HIGH-PERFORMANCE EVOLUTIONARY COMPUTATION FOR SCALABLE SPATIAL OPTIMIZATION

BY

YAN LIU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Informatics
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Professor Shaowen Wang, Chair
Professor Ximing Cai
Professor Wendy K. Tam Cho
Professor Sara McLafferty

ABSTRACT

Spatial optimization (SO) is an important and prolific field of interdisciplinary research. Spatial optimization methods seek optimal allocation or arrangement of spatial units under spatial constraints such as distance, adjacency, contiguity, partition, etc. As spatial granularity becomes finer and problem formulations incorporate increasingly complex compositions of spatial information, the performance of spatial optimization solvers becomes more imperative. My research focuses on scalable spatial optimization methods within the evolutionary algorithm (EA) framework. The computational scalability challenge in EA is addressed by developing a parallel EA library that eliminates the costly global synchronization in massively parallel computing environment and scales to 131,072 processors. Classic EA operators are based on linear recombination and experience serious problems in traversing the decision space with non-linear spatial configurations. I propose a spatially explicit EA framework that couples graph representations of spatial constraints with intelligent guided search heuristics such as path relinking and ejection chain to effectively explore SO decision space. As a result, novel spatial recombination operators are developed to handle strong spatial constraints effectively and are generic to incorporate problem-specific spatial characteristics. This framework is employed to solve large political redistricting problems. Voting district-level redistricting problems are solved and sampled to create billions of feasible districting plans that adhere to Supreme Court mandates, suitable for statistical analyses of redistricting phenomena such as gerrymandering.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to thank my thesis committee members, Ximing Cai, Wendy K. Tam Cho, Sara McLafferty, and Shaowen Wang for their advice and comments. My dissertation journey has been a unique one and leaves a trail of gratitude. My advisor, Shaowen Wang, supported my pursuit of the doctoral degree in Informatics while working full-time in his lab. Professor Cho graciously gave her time in discussing and advising my thesis research. Professor Cai and Professor McLafferty generously shared insightful thoughts on the coupling of GIS and operations research.

My initial research career started at The University of Iowa. I would like to thank Professor Sukumar Ghosh, Alberto M. Segre, Hantao Zhang, Ted Herman, and the Department of Computer Science for their generosity and open mind that allowed me to grow.

Personally, I would like to thank Min Zhang, Chenyuan Qi, Heqing Shi for their consistent encouragement and support, without which the completion of this thesis would not have been possible.

Finally, I thank the University of Illinois at Urbana-Champaign for providing me with a supportive academic environment. The National Center for Supercomputing Applications provided me with a spacious office that I utilized daily for my work. Karin Readel exhibited enormous care in tending my progress in the Ph.D. program at the Illinois Informatics Institute. I am also grateful for the wonderful staff and students at the CyberGIS Center, who provided fun, encouragement, and great working relationships.

I dedicate this thesis to my father and my late mother for their endless love and trust.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Motivation 2
1.2 Research Framework 4
1.3 Thesis Overview 7

CHAPTER 2 SCALABLE PARALLEL EVOLUTIONARY ALGORITHMS 14
2.1 Literature Review 17
2.2 Problem of Study 19
2.3 Parallel Genetic Algorithm Design 20
2.4 Evaluation 32
2.5 Summary 59

CHAPTER 3 TRANSFORMING POLITICAL REDISTRICTING THROUGH SCALABLE EVOLUTIONARY COMPUTATION 61
3.1 Redistricting 62
3.2 Literature Review 64
3.3 Redistricting Analysis: A New Computational Approach 67
3.4 Evolutionary Algorithm 72
3.5 Parallelization 81
3.6 Evaluation 83
3.7 Conclusion and Discussion 98

CHAPTER 4 SPATIALLY EXPLICIT EVOLUTIONARY COMPUTATION 99
4.1 Literature Review 101
4.2 Methodology 104
4.3 Algorithms 115
4.4 Evaluation 123
4.5 Conclusion and Discussion 131

CHAPTER 5 CONCLUSION AND FUTURE DIRECTIONS 135
5.1 Summary of Contributions 135
5.2 Discussion and Future Work 138

REFERENCES 144

CHAPTER 1

INTRODUCTION

In the discipline of geography, spatial optimization has been an important and prolific research field. Spatial optimization methods seek to allocate or arrange spatial units in a way that optimizes some objective or measure of goodness. Optimally arranging spatial units has many applications in location and allocation, coverage, planning, zoning, etc. As spatial granularity becomes finer and problem formulations incorporate increasingly complex compositions of spatial information, the performance of spatial optimization solvers becomes more imperative. A solver must be able to scale to problem size and complexity to obtain optimal or near optimal solutions. My research focuses on the computational challenges presented by developing scalable and effective heuristics for solving large-scale spatial optimization (SO) problems within the framework of evolutionary algorithms (EA).

As problem size increases, the decision space defined by problem variables often explodes exponentially because many combinatorial optimization problems are *NP*-hard (Garey and Johnson, 1979). Problems with spatial objectives and constraints further complicate the solution landscape in the decision space. Such spatial correlation, often non-linear, significantly affects the performance of decision space search by solvers if they do not incorporate spatial elements appropriately in search strategies. I tackle the scalability challenge of EA for spatial optimization from the following three perspectives.

- **Computational scalability**. Apart from problem-specific knowledge, one route for pursuing a general scaling strategy is to exploit the massive computing power that is available via high-performance computing architecture. Desirable scalability can be achieved simply by employing more compute processors. This strategy requires careful consideration of how to coordinate inter-processor communications among a large number of processors to enable efficient cooperative optimization in EA.

- **Limitations of classic EA operators in solving large spatial optimization problems**. The effectiveness of classic EA is heavily reliant on randomization linear recombination operators such as crossover and mutation (Holland, 1992). However, linear recombination for finding feasible and better solutions in a decision space

constrained by non-linear spatial configurations is ineffective, especially in large-scale spatial optimization.

- **Spatially explicit EA operators**. Many spatial optimization applications incorporate spatial constraints and objectives at the problem formulation phase. However, once formulated, a solver's iterative search process, i.e., the traversal of decision space, is often conducted without awareness and guidance of spatial characteristics. Incorporating spatial elements as problem-specific knowledge in defining neighborhood functions, a building block for various search heuristics, thus becomes critical. How to couple them in heuristic strategies for decision space search is, however, a challenge.

1.1 Motivation

The rise in the size and richness of problem data poses new challenges and research frontiers because there are now both a larger number of problem variables and additional objectives and constraints that need to be incorporated in problem formulation. A combinatorial optimization problem is often *NP*-hard, i.e., computationally intractable. There are simply more possibilities, leading to a rapidly rising decision space size that quickly eclipses the capabilities of exact algorithms to identify optimal solutions. In the spatial optimization realm, for instance, examples of large applications are becoming commonplace, marching alongside the need for more impactful and comprehensive spatiotemporal decision making scenarios using input data of high-resolution and broad coverage. Examples include political redistricting at the voter tabulation district or census block level (Liu, Cho and Wang, 2015), maximum coverage for hundreds of demand sites and tens of thousands of facility sites (Tong, Murray and Xiao, 2009), and grid-based optimization on large numbers of grid cells for high-resolution raster input.

Solving very large spatial optimization problems requires a careful examination of two major facets of a solver: *effectiveness*, i.e., how to navigate the search of decision space intelligently to find better solutions, and *efficiency*, which asks how to conduct performant search numerically and algorithmically. The former has been the foci of operations research. As most of spatial optimization applications deal with discrete spatial units, a spatial optimization problem is often a combinatorial optimization problem (at least partially). It has been an active area of research with the development of many effective exact and heuristic search methods (e.g., local search, LP/ILP, mathematical relaxation, simulated annealing, scatter search, path relinking, tabu search, and population-based heuristic algorithms such as evolu-

tionary algorithms, ant colony, and particle swarm) (BoussaïD, Lepagnot and Siarry, 2013). Advances of these optimization methods have been driven by improvements in algorithms, mathematical modeling, and computing technologies.

I am interested in one class of broadly adopted heuristic algorithms: evolutionary algorithms (EA). Inspired by natural selection, Genetic Algorithms (GA) and its more general form, Evolutionary Algorithms (EA), comprise a generic heuristic method for finding near-optimal or optimal solutions to difficult search and optimization problems (Holland, 1992). EA mimics the iterative biological evolution and starts with a set of solutions encoded into a population. Through GA operators (e.g., selection, crossover, mutation, and replacement) that are often stochastic, the population evolves based on the "survival of the fittest" rule (Wright, 1932; Goldberg, 1989). Such an evolutionary process stops when the population converges to solutions of specified quality. The theory behind EA has been extensively studied and shown to be a generic and effective heuristic for solving non-linear optimization problems. The applications of EA theory in developing specialized solvers and solving particular optimization problems are abundant for small-sized problems.

The efficiency consideration is, on the other hand, a computation problem. While developing numerically efficient solvers is one solution, exploiting more computing power is more scalable. But it often requires the development of parallel heuristic algorithms. Handling parallel computing in heuristic algorithms has been challenging due to the difficulty in efficiently coordinating the entire problem solving process across a large number of processors. However, recent advances in high-performance computing (HPC) now present unprecedented opportunities for algorithms to leverage the computing power provided by supercomputing on national cyberinfrastructures at extreme scales. For example, supercomputers are marching toward the exascale computing era in which computing power at the exaflop (10^{18} numerical calculations per second) level can be used in a single problem-solving computation. Such enormous computing power is provided by millions of processors interconnected via low-latency and high-bandwidth network at a supercomputer center. The aforementioned challenge in developing parallel algorithms becomes prominent to tackle in order to exploiting millions of processors in optimization computation. This is, fundamentally, a computational scalability problem in algorithm design that has been identified as one of the top 10 challenges in exascale computing (Lucas et al., 2014). In turn, scalable computation solutions have significant impact on heuristic algorithm research similar to how they transform other science domains (Ashby et al., 2010). An immediate benefit, of course, is the capability to solve larger optimization problems by simply adding computing power from supercomputers.

In this thesis, the term "scalability" thus refers to the scalability to both problem size and the number of processors.

Above observations drive my research to develop a scalable EA framework for spatial optimization at large scale. By coupling computation, heuristics, and spatial optimization together, I developed a research agenda to investigate fundamental challenges in existing EA approaches and propose a novel EA framework for scalable evolutionary computation.

1.2 Research Framework

"All things are related, but nearby things are more related than distant things."
– Waldo Tobler on the First Law of Geography

My thesis focuses on extreme-scale evolutionary computation for spatial optimization. While spatial elements make problem-solving more challenging, it also offers an avenue for innovative solutions if EA are spatially cognizant. In particular, I incorporated *spatial thinking* (Goodchild and Janelle, 2010) to couple the three research pillars in this thesis: parallel evolutionary computation, spatially explicit EA operators, and large spatial optimization applications. **First, I focus on resolving major computational bottlenecks in parallel evolutionary algorithms (PEA) that prevent desirable scalability for large numbers of processors.** A PEA employs a set of computing processors spatially connected via a network topology. Identifying computing and communication bottlenecks on these spatially districted processors through performance profiling and analysis of the parallelisms in PEA yields insight into the algorithmic behavior and the scalability of PEA, which in turn help us design computationally scalable PEA solutions. **Second, I build scalable spatial optimization applications to demonstrate the performance enhancements.** These applications enable the study of how the enhanced scalability can also help expand the understanding of application-specific insights. In addition, closely examining large applications reveals limitations of classic EA operators. **Third, a spatially explicit EA framework is proposed to significantly improve the performance of EA in large spatial optimization**. In particular, I design novel general spatial EA operators that strategically direct neighborhood search, in addition to randomization, to effectively explore vast plateaus of the decision space constrained by spatial configurations. These operators overcome the drawback of linear recombination-based EA operators.

Spatial thinking is the cornerstone of research methodologies in many science domains, especially and obviously, in geography and geographic information science (Downs et al.,

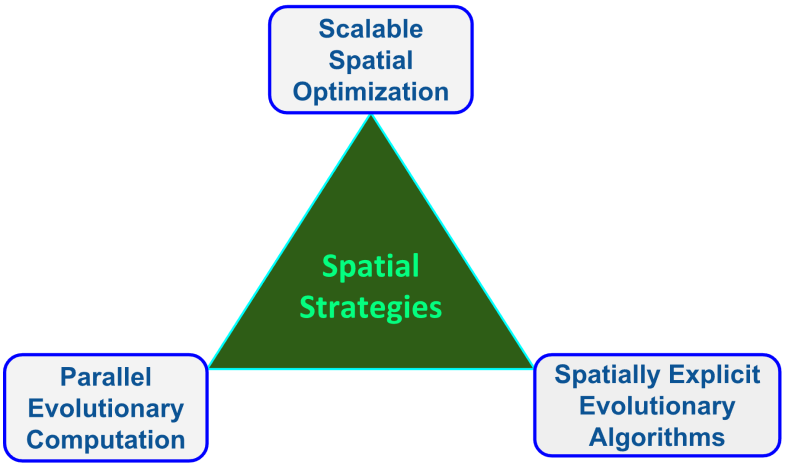


Figure 1.1: Research framework.

2006). In geospatial analysis, modeling, and simulation, a spatialization process constructs abstract spaces for studying insights of a problem that is, otherwise, not easy to model. A model is deemed *spatially explicit* when it differentiates behaviors and predictions according to spatial location and associated spatial properties (Goodchild and Janelle, 2010).

Figure 1.1 illustrates my overarching research framework. A central theme that connects the work on the three research foci in EA algorithms, computation, and applications is, interestingly, the **spatial** abstraction along three orthogonal dimensions of evolutionary computation in spatial optimization (SO): the *spatial configuration of PEA topology*, *decision space* and the spatial trajectory of neighborhood search, and the *incorporation of SO elements in the traversal of the decision space*.

Spatial configuration of PEA topology. To achieve desirable computational scalability, appropriate spatial configuration of participating computing processes is an integral component of PEA design. Any parallelization of EA (e.g., either fine-grained and coarse-grained PEAs (Liu and Wang, 2015)) assumes a particular topology of EA processes. Each EA process hosts one or more solutions, termed *local population*. The structure of a local population is sometimes exploited in the EA selection operator to influence the selection of parenting solutions. Collectively, local populations form a global population, and solutions from a local population travel to neighboring EA processes through migration. The topology and communication among direct neighbors are supported through hardware (e.g., the fine-grained PEA on SIMD machines) or PEA operators such as migration. Solution propagation among EA processes is done via broadcasting or by hop-by-hop communication. In either case, on a square grid topology, a solution takes at most $O(\sqrt{n})$ hops, which is also the

diameter of the topological space, to reach to all the EA processes, where n is the number of EA processes. As such, other communication parameters and statistics in PEA can be computed by using a 2D cartesian coordinate system, which is spatial. Such a spatial configuration makes it possible to study PEA operator design and its scalability to the number of processors in our work described in chapter 2.

Decision space and the spatial trajectory of neighborhood search. Let us consider the spatialization of decision variable space in combinatorial optimization. A general form of a combinatorial optimization problem can be described as

$$\text{minimize } C(x) : x \in X \text{ in } R_n, \text{ such that } A_i x \leq b_i, \ i = 1, \ldots, p \tag{1.1}$$

In this formulation, x, the problem variable vector of size n, has a specific subset of discrete variables. $C(x)$ and $A_i(x)$ are objective and constraint matrices, respectively. The combinations of X comprise the decision variable space, (in short, *decision space*). We define *search space* as the surface composed of all solutions, feasible and infeasible, that EA operators iteratively identify. Each solution is a point on the surface. This surface is often referred to as the solution landscape (Tayarani-N and Prügel-Bennett, 2014). In heuristic algorithms, The exploration of the search space for identifying new solutions is often via a neighborhood search function, termed *neighborhood function*. A common tradeoff in neighborhood function design is how far the search radius can be versus how much computing cost is needed to reach the defined search radius. This tradeoff determines the gap between the local and global knowledge of the solution landscape.

The decision space and the search space depict the spatial trajectory of a neighborhood search in a heuristic algorithm. In general, heuristic algorithms have two common components: a component that ensures that the search conducted by the algorithm converges, and a neighborhood search process that produces new solutions. The neighborhood search process constructs a search neighborhood graph, in which two points are connected if the search path goes from one to the other. In an EA's crossover operator, new solutions are generated from two parent solutions. The distance, which can be defined as the number of bits assigned with difference values in two parent solutions, can be used to define the various linear combination strategies in classic crossover implementations such as the linear crossover (Wright et al., 1991), the arithmetical crossover (Michalewicz and Janikow, 1991), and the average crossover (Davis, 1989). Based on this distance, moving from one parent solution to the other creates a set of paths, along which new intermediate solutions are generated and leveraged, if better. This is the general search principle of the path relinking

search heuristic (Glover, 1994). The mutation in EA can also be considered a neighborhood function, defined on a single parent solution with the neighborhood defined as the candidate solutions obtained by mutating one or several bits in x. The spatialization of EA recombination operations in favor of strategically directed search for variable interdependency led to the definition of spatial properties (e.g., distance, search radius, search preference regions, and paths) in our EA recombination operator design, described in chapter 4.

Spatial optimization (SO) elements. Variable interdependency in the decision space is represented in the objectives and constraints and the numerical relationships among variable values in x. Such variable interdependency is hidden but known problem-specific knowledge that, if a solver can exploit effectively, can help accelerate the optimization process dramatically. For example, the n-opt local search heuristic in Helsgaun's TSP solver is a generic search heuristic that is extremely effective in incrementally finding better tours by exploring potential edge space around the vertices of n pairs of selected edges (Helsgaun, 2000), followed by a Lagrangian relaxation process to optimize the edge combination based on edge weight distribution. The degree to which problem insights can be exploited by an algorithm varies dramatically between different problems. For example, one can exploit the relationship between weights and values to enact effective dynamic programming algorithms for the single knapsack problem. However, multiple knapsack problems, such as the Generalized Assignment Problem (GAP), are much more difficult because the weight-value relationship is complicated with multiple knapsacks, exhibiting weak variable interdependency.

Spatial optimization, where problem variables are spatial units (points, lines, or polygons) and objectives and constraints are formulated through spatially explicit definition of spatial relationships (i.e., topological or geometric relationships), represents a class of promising problems for EA to explore effective neighborhood search strategies from the underlying spatial characteristics. Capturing spatial elements (e.g., distance, adjacency, contiguity, containment, intersection, shape, partitions, and pattern (Tong and Murray, 2012)) can significantly accelerate neighborhood search performance, which is critical for traversing decision space of large problems. At an abstract level, spatial thinking serves as a unifying framework in my research to conceptualize the scalability study in computation, neighborhood search in EA, and SO-specific EA.

1.3 Thesis Overview

For each of the three research foci depicted in Figure 1.1, the corresponding scientific study follows a four-step process to identify major computational and algorithmic challenges, pro-

pose models and methodologies to formulate research problems, design algorithms, and conduct performance evaluation, respectively. Through this process, the following work has proceeded.

1.3.1 Building a Scalable PEA Library

A scalable computation solution is developed to provide a high-performance PEA library in order to tackle the computational scalability challenge on supercomputers. The following research questions were identified and pursued to guide the development of this research:

Q1: What are the most significant computational bottlenecks in exploiting massively parallel computing architecture to achieve scalable PEA?

Q2: How can we devise high-performance PEAs to eliminate such computational bottlenecks?

The most recent advances in high-performance computing (HPC) move toward exascale computing (Ashby et al., 2010) and provide a promising direction toward scalable evolutionary computation. In identified top challenges in exascale computing, developing exascale algorithms is one of them (Lucas et al., 2014). While most of the existing research on PEAs are developed and tested on a single computer in multi-threaded fashion, my work focuses on scalability on a high-performance computer with massively parallel architecture. Ideally, PEA software is able to scale by simply adding more computing nodes. Auspiciously, EA, known as an effective heuristic for finding optimal or near-optimal solutions to difficult optimization problems, has an inherently parallel structure for exploiting high performance and parallel computing resources for randomized iterative evolutionary computation.

At the same time, it remains a significant challenge to devise parallel genetic algorithms that are able to scale to massively parallel computer architecture (also known as the mainstream supercomputer architecture) primarily because: (1) a common PEA design adopts synchronized migration, which becomes increasingly costly as more processor cores are involved in global synchronization; and (2) asynchronous PEA design and associated performance evaluation are intricate because PEAs are stochastic in nature and the amount of computation work needed to solve a problem is not simply dependent on the problem size.

To address these challenges, chapter 2 describes my present work in developing a scalable coarse-grained PEA library (PGAP) and the experience of solving a well-known *NP*-hard optimization problem, the Generalized Assignment Problem (GAP). Specifically, an asyn-

chronous migration strategy is developed to enable efficient deme interactions and significantly improve the overlapping of computation and communication. Breaking the synchronization barrier, however, introduces numerous tasks for managing the communication and coordination of PEA process interactions efficiently and correctly at both the application and system level. A major issue with buffer overflow arises here. I investigate the relationship of buffer overflow with migration parameters in order to resolve observed message buffer overflow and the loss of good solutions obtained from migration. Two algorithmic conditions were then established to detect these issues caused by communication delays and improper configuration of migration parameters and, thus, guide the dynamic tuning of PEA parameters to detect and avoid these issues.

EA is a class of stochastic algorithms. The stochastic nature and the consequent variation in computing cost and output results present a challenge for performance evaluation. In this work, methodologies for analyzing the scalability and numerical performance of the developed PEA library are discussed. A comprehensive performance evaluation scheme is proposed to appropriately compare a set of measurements for better understanding both the algorithmic and computational performance of the library.

The work on this scalable PEA/PGA library has been published (Liu and Wang, 2015). The major contributions of this work include:

- A novel asynchronous migration strategy is developed to achieve high scalability in PEA.
- A scalable PEA is developed for the *NP*-hard GAP problem.
- The algorithmic analysis resolves the buffer overflow issue in asynchronous PGAs and can be used as guideline to configure PEA parameters.
- The strong and weak scaling tests demonstrate the superiority of our PGA approach.
- This is the first PEA library, to the best of our knowledge, which scales up to 131,072 processors.
- The PEA library can be used as a generic EA solver with minimum modification.

1.3.2 Solving Large Spatial Optimization Problems

Built on a scalable EA library for spatial optimization, we present the political redistricting application, a spatial optimization problem in political science and geographic information

science known for a comprehensive set of spatial elements in its problem formulation, as a case study. The following research questions were pursued when applying the EA approach to large redistricting instances.

Q3: How does one incorporate explicit and implicit spatial constraints and objectives, transformed from redistricting problem domain, for large redistricting problem-solving?

Q4: How can we enable statistical analysis on redistricting, which requires massive numbers of high quality district maps?

The problem of political redistricting can be formulated as a spatial optimization problem, with objectives and constraints defined to meet legal requirements. The formulated optimization problem is *NP*-hard. Our scalable evolutionary computational approach solves redistricting as a representative spatial optimization application by incorporating strong substantive knowledge and keeping deep adherence to Supreme Court mandates. Since the spatial configuration plays a critical role in the effectiveness and numerical efficiency of redistricting algorithms, EA operators exploit explicit and implicit spatial characteristics in redistricting requirements and effectively search the variable space. The scalable computation framework of our library is coupled with a sampling strategy to create billions of *reasonably imperfect* solutions suitable for statistical analysis of a set of redistricting measures.

This research application has been published with case studies of North Carolina and Maryland (Liu, Cho and Wang, 2016; Cho and Liu, 2016). The major contributions of this work include:

- A scalable evolutionary computational approach to political redistricting optimization.
- Effective and efficient EA operators for handling spatial constraints and objectives.
- Our approach enables substantive redistricting research and practice at large scale.
- The case study on North Carolina and Maryland demonstrated desirable problem-solving capabilities that can directly assist decision-making scenarios in court and open research as a computational solution.

1.3.3 Developing Effective Search Strategies for Spatial EA

Adapting heuristics to function efficiently with spatial elements has been a consistent effort in geographic analysis research. Many existing efforts have focused on representing spatial constraints and objectives in SO problem formulation. Most of the SO problem-solving practice

utilize general optimization solvers (e.g., LP/IP solvers, general EA software). The drawback of this practice is that spatial insights such as dependency and heterogeneity influence the problem-solving process only at evaluation stage of each iteration of the optimization rather than being incorporated directly into search operators, which introduce majority of the computation cost. My goal is to develop a generalization of these basic search operators that are effective in traversing vast decision space for large problems and are capable of explicitly incorporating spatial elements into neighborhood search. I pursue the following two research directions.

Q5: What are major limitations of classic EA recombination operators (e.g., crossover and mutation) in solving spatial optimization problems?

Q6: How to develop a constructive spatial recombination framework that is capable of traversing spatially constrained decision space for the effective generation of new solutions?

To answer these questions, I propose that *EA recombination operators must be not only spatially aware, but also spatially explicit in exploring the spatially constrained decision space.* To achieve the spatial explicitness, *constructive methods* are more promising than repair methods, which is particularly the case when problem size is large. By investigating existing spatial EA solutions and studying the requirements for effective spatial recombination, I propose a novel spatially explicit evolutionary computation framework as a solution to this challenge. Specifically, the following research contributions are made:

- A major limitation of classic EA in capturing spatial characteristics of SO problems is identified. Embedding spatial characteristics within the crossover and mutation operation is important, but missing.
- A novel spatially explicit evolutionary computation framework is proposed to implement spatial recombination through guided search heuristics of path relinking for crossover and ejection chain for mutation.
- The proposed framework is the first, to the best of our knowledge, EA that employs effective path relinking and ejection chain heuristics for spatial recombination.

The thesis is organized as follows. Chapter 2 describes the design of a scalable PEA and associated computational challenges, solutions, and performance evaluation intricacies.

Chapter 3 presents a scalable evolutionary algorithm for a large-scale spatial application—political redistricting. The computational advantage of my solutions and effective heuristics designed for EA operators enable the practical use in the optimization and statistical analysis of redistricting in the United States. New insights into the redistricting process are revealed through the analysis. Our approach has promising potential for transforming the redistricting process in the future. Chapter 4 proposes a novel design for EA crossover and mutation algorithms tuned for large problem solving. The proposed EA operators are evaluated by their ability to solve spatial optimization problems that exhibit strong variable interdependency in spatial objectives and constraints.

1.3.4 Terminology Notes

Evolutionary algorithms (EA) is a general term to describe the class of population-based iterative optimization algorithms that follow the basic evolution "survival of the fittest" principle. Genetic algorithm (GA) often refers to a subset of EA that encodes a problem as chromosome—a linear string of problem variables. In GA, classic crossover and mutation are coded in a fashion that closely mimics their counterpart in evolutionary biology, albeit in bit operations on binary strings. In this thesis, these two terms are used interchangeably because the discussion on the recombination operators is not restricted to the classic crossover and mutation. Similarly, parallel EA (PEA) and parallel GA (PGA) are used interchangeably. Processor, core, or the combined form "processor core" are used interchangeably to denote a physical CPU that runs one process in parallel computing environment.

The term "scale" has specific definition in computational science and geography, respectively. The computational scale refers to the computational intensity and complexity impacted by problem size and the amount of employed computing power. Large scale computation tends to use more computing power for larger problem size. The geographical scale defined in GIScience and cartography refers to the mapping scale, where large scale maps delivers small geographic coverage but with high resolution. In this thesis, I adopt the definition from computational science for the convenience of description. When geographic scale needs to be mentioned, I will spell out the term and clarify.

We highlight a few terms that are common to both operations research and geographic analysis, and that are central to our work. In particular, a general spatial optimization problem optimizes an objective function cx, where c is a cost matrix and x is the decision variable vector. The entire x or a subset of it contains spatial variables. This is solved by satisfying a series of constraints $Ax \leq b$, where b is the constraint threshold and A is

the coefficient matrix. We are only interested in problems where x or at least part of x is comprised of integer values. A few definitions follow.

- *Space*. The term *space* in this thesis refers to *decision space* formed by decision variables and their value ranges. *Search space* is a subset of the decision space explored by a heuristic algorithm or function.
- *Path*. The term *path* in this thesis refers to a search path in the decision space.
- *Spatial*. The term *spatial* in this thesis is defined in geographic context only.
- *Neighborhood*. The term *neighborhood*, or *locality*, can be used to describe both the decision and geographic space, with appropriate clarification. Two neighboring solutions in the decision space may not be spatially proximate.

The term *spatial variable* and *spatial unit* are interchangeably used in this thesis. Both refer to basic discrete spatial object that is not divisible further in problem definition and appears as a problem variable in mathematical formulation of a SO problem.

CHAPTER 2

SCALABLE PARALLEL EVOLUTIONARY ALGORITHMS

Inspired by natural selection, Genetic Algorithms (GA) and its more general form, Evolutionary Algorithms (EA), comprise a generic heuristic method for finding near-optimal or optimal solutions to difficult search and optimization problems (Holland, 1992). GA mimics iterative evolutionary processes with a set of solutions encoded into a population at the initialization stage. Through GA operators (e.g., selection, crossover, mutation, and replacement) that are often stochastic, the population evolves based on the "survival of the fittest" rule (Wright, 1932; Goldberg, 1989). Such an evolutionary process stops when the population converges to solutions of specified quality. The computational challenges of GA are attributed to both problem-specific characteristics (e.g., problem "difficulty" (e.g., *NP*-hard), problem size, the complexity of fitness function, and distribution characteristics of solution space specific to problem instances), and runtime efficiency of stochastic search (Oliveto, He and Yao, 2007).

High performance and parallel computing has been extensively studied to tackle the aforementioned computational challenges in the framework of GAs since this framework has inherent parallelism embedded in the evolutionary process (Alba and Tomassini, 2002). For example, a population can be naturally divided into a set of sub-populations (also called demes) that evolve and converge with a significant level of independence. Various types of parallel genetic algorithms (PGAs) have been developed and broadly applied in a rich set of application domains (Alba and Tomassini, 2002; Konfrst, 2004; Huang and Rajasekaran, 2004; Hidalgo et al., 2010). More interestingly, previous work by Alba and Troya (1999*b*) showed that PGA computation not only improves computational efficiency over sequential GAs, but also facilitates parallel exploration of solution space for obtaining more and better solutions. In fact, Hart et al. (1996) showed that running PGA even on a single processor core outperformed its sequential counterpart. As a result, PGA is often considered and evaluated as a different algorithm rather than just the parallelization of its corresponding sequential GA.

This chapter describes a scalable PGA solution to exploiting massively parallel high-end computing resources for solving large problem instances of combinatorial optimization problems. The proposed PGA approach focuses on the scalability to massively parallel processor cores (referred to as cores hereafter) available from high-end computing resources such as those provided by the National Science Foundation XSEDE (*XSEDE*, 2017) cyberinfrastructure. The algorithm is a coarse-grained steady-state (Alba and Troya, 2002) PGA that searches the solution space in parallel based on independent deme evolution and periodical migrations among connected demes. Scalability is key to efficiently exploiting a large number of cores in parallel.

Previous PGA implementations mostly rely on synchronization to coordinate parallelized operations (e.g., the migration operation in coarse-grained PGAs and the selection operation in fine-grained PGAs), primarily because the computation of PGA is an iterative process and it is straightforward to implement iteration-based synchronization. Synchronization is often needed at two places. The first is while waiting for all PGA processes to rendezvous before migration operations. The second is while using synchronous communication to exchange data. While success in scaling PGA to many cores has been achieved based on hardware instruction-level synchronization supported by SIMD architectures (Shapiro, Wu and Bengali, February 2001), we argue that the computational performance of PGA is under-achieved through synchronizing iterations across massively parallel computing resources with MIMD architecture (McCool, 2008).

Accordingly, we pursue an asynchronous migration strategy that is designed to achieve scalable PGA computation through a suite of non-blocking migration operators (i.e., export and import) and buffer-based communications among a large number of demes connected through regular grid topology. The asynchrony of migration effectively removes not only the costly global synchronization on deme interactions, but also allows for the overlapping of GA computation and migration communication.

Addressing buffer overflow issues caused by inter-processor communications and understanding their relationship to the configuration of asynchronous PGA parameters are crucial for the design of scalable PGA on high-end parallel computing systems. Through algorithmic analysis, we identified two buffer overflow problems in exporting and importing migrated solutions, respectively. In export operations, the overflow of the outgoing message buffer used by the underlying message-passing library may cause runtime failure and cause the PGA computation to abort. In import operations, the overflow of the import pool maintained for receiving solutions from neighboring demes may cause the loss of good solutions. Through

theoretical analysis, we derive two conditions to guide the setting of PGA parameters, including migration parameters, topology, and buffer sizes based on the underlying message passing communication library in order to detect and/or avoid these aforementioned buffer overflows. To the best of our knowledge, our work is the first to explicitly consider the relationship between the configuration of asynchronous PGA parameters and underlying system characteristics to improve the reliability of asynchronous PGAs.

Experimental results showed that, with the asynchronous migration strategy, our scalable PGA is able to efficiently utilize 16,384 cores with significantly reduced communication cost. Specific strong and weak scaling tests were designed to evaluate the scalability and numerical performance of PGA because conventional weak and strong scaling methods are not directly applicable to PGA. For example, the problem size used in conventional weak scaling is not a good indicator of the amount of computation needed to achieve a certain solution quality in PGA. PGA and sequential GA also have different algorithmic behaviors. More importantly, population size plays a crucial role in PGA performance as the number of cores increases. Therefore, instead of problem size, we chose population size to study PGAP scalability and compare its performance with the corresponding sequential algorithm. In strong scaling tests, linear and super-linear speedups were observed in solving large Generalized Assignment Problem (GAP) instances. Our PGA also outperforms the best-so-far sequential GA in solving modest problem instances. Studies of numerical performance on large instances exhibited promising results for the capability of the PGA to exploit massive computing power to more effectively explore the search space and identify alternative solutions, converge faster, and obtain improved solution quality.

The PGA solution was further extended as a PGA library and was extensively evaluated on multiple supercomputers at multiple scales, up to 131,072 cores, including Blue Waters, the most powerful supercomputer in the world in an academic research computing environment. The PGA library's performance was also tested on a heterogeneous supercomputing architecture comprised of host processors and co-processors on the Stampede supercomputer. Results on Stampede show desirable resilience to the processing capability difference among heterogeneous CPUs.

The reminder of the chapter is organized as follows. Section 2.1 reviews related work on PGA. Section 2.2 introduce the study problem, the GAP problem. Section 2.3 describes the design and implementation of PGA for solving computationally intensive GAP instances. Section 2.4 details computational experiments designed for evaluating PGA scalability and

numerical performance and analyzes experiment results. Section 2.5 summarizes the research findings.

2.1 Literature Review

PGAs are well-known approaches for increasing the computational efficiency of GA. The parallelization can be exploited in various ways. *Fine-grained* PGAs (Whitley, 1993) (also referred to as *cellular GA (cGA)* or *Diffusion Models* (Eklund, 2004)) parallelize the selection operator to select parents from directly connected neighbors on a PGA topology in each iteration. *Coarse-grained* PGAs (also referred to as *island model GA (iGA)*) migrate a portion of local solutions to connected demes periodically (Ruciski, Izzo and Biscani, 2010). *Global parallelization*-based PGAs parallelize the computation of the fitness evaluation function if the function is computationally intensive (DAmbrosio and Spataro, 2007). PGA surveys can be found in literature (Belding, 1995; Whitley, 2001; Rivera, 2001; Alba and Troya, 1999*b*; Alba and Tomassini, 2002; Cantu-Paz, 1997). Tanese (1989) showed that a PGA exhibits different algorithmic behavior than its corresponding sequential GA.

Interestingly, even without communication, PGAs could improve GA performance with independent deme evolution. This is because multiple independent running instances of the same GA likely produce results of differing quality simply due to the randomness introduced by each GA instance. Given a fixed solution quality requirement, the probability that at least one instance finds a solution of designated quality earlier increases with the number of cores utilized. Even without communications among demes, therefore, running multiple demes on a single core often performs better than running a single deme with the same global population size. On the other hand, communications, i.e., migration of a portion of local solutions to other demes, greatly improves the effectiveness of GA computation through the propagation of good solutions and the injection of new randomness.

Previous work on designing scalable PGAs often employs the fine-grained PGA model on SIMD architecture (Shapiro, Wu and Bengali, February 2001; Chen, Flann and Watson, 1998; Kalinowski, 1994; Baluja, 1992; Juille and Pollack, 1996; Prabhu, Buckles and Petry, 2006; Ngo and Marks, 1993). For example, Shapiro, Wu and Bengali (February 2001) implemented a PGA for RNA folding on the MasPar SIMD supercomputer with 16,384 processing units. Chen, Flann and Watson (1998) developed a hybrid cellular GA on the same machine. These fine-grained PGA implementations exhibit good performance on gene diffusion, i.e., the propagation of high-quality genes across the entire population. Hart et al. (1996), Alba and Troya (2002), and Prabhu, Buckles and Petry (2006) pointed out that SIMD architecture

is suitable for the development of fine-grained PGAs because fine-grained PGAs require synchronization in each iteration to select parents from directly connected demes and SIMD systems provide hardware- and/or system-level synchronization support that can be directly adopted to coordinate the selection operation within each iteration.

It is more challenging to synchronize GA iterations on MIMD systems in which processors are often connected through a network. On MIMD architecture, synchronization can be a costly task at the system level, especially when it involves a large number of connected cores. In message passing models where MIMD systems are often employed for communication, the overhead of synchronization has been well studied (Jiang et al., 2004; Faraj, Patarasuk and Yuan, 2007). Such synchronization cost can cause serious performance degradation on PGAs. In the synchronous mode, a PGA iteration can be considered as a sequence of computations followed by a global barrier to rendezvous all processes. Any delay at one process, caused from various sources (e.g., network, operating system, memory, I/O, or computation itself in GA) is propagated to all of the participating processes, and the probability and duration of such delay increases as the number of processes increases. In fact, the evolutionary process in PGA may not require synchronization because: (1) a PGA process can evolve independently; (2) communications with other processes only involve neighboring processes, without the need for global-level synchronization; and (3) sending information to and receiving information from neighboring processes does not need to be coupled.

In general, GA, as a type of stochastic computing model, exhibits massive parallelism and holds tremendous potential for reaping the benefits of large-scale parallel computation (Cledat et al., 2009). As massively parallel computing resources become available (McCool, 2008; Asanovic et al., 2006) as a consequence of active development of multi-core/many-core computing and extreme-scale supercomputing (Bader, 2007), it is promising to harness an unprecedented amount of parallel computing resources for scalable GA computation.

It has been recognized that asynchronous inter-deme interactions are desirable for designing scalable PGAs (Alba and Troya, 1999*a*, 2002; Hart et al., 1996; Lin, Punch and Goodman, 1994). Most of the previous PGA work chose to synchronize GA computation primarily for the purpose of straightforward parallel programming implementation. Early work by Hart et al. (1996) showed that delays exhibited in asynchronous fine-grained PGAs helped achieve better solution quality because some demes were allowed more time for the local evolution to be more convergent toward better solutions. Lin, Punch and Goodman (1994) observed the effectiveness of exchanging solutions asynchronously and proposed a solution exchange strat-

egy based on population similarity instead of the fixed connection topology on coarse-grained PGAs. The CAGE framework (Folino, Pizzuti and Spezzano, 2003) employed non-blocking message passing functions for the development of fine-grained GA programming library on MIMD clusters configured with up to 256 computing nodes. We argue that the availability of massively parallel computing resources now offers a major driver toward the design and development of asynchronous PGAs. In this paper, an asynchronous migration strategy is developed as the mainstay for enabling scalable PGA.

2.2 Problem of Study

In order to conduct an in-depth study of PGA performance, we study the implementation of an algorithm for Generalized Assignment Problem (GAP), a well-known combinatorial optimization problem. The design of the PGA components is problem-agnostic, although the PGA solver for GAP is named PGAP.

The GAP problem is to find an optimal assignment of n items to m bins (knapsacks) such that the total cost of the assignment is minimized and the weight capacity constraint of each bin is satisfied. The cost and weight of an item depend on both the item itself and the assigned bin (Borndorfer and Weismantel, 1997). GAP belongs to the class of *NP*-hard 0-1 Knapsack problems (Fisher, Jaikumar and Wassenhove, 1986; Chekuri and Khanna, 2000; Borndorfer and Weismantel, 1997). It is the generalization of the well-known Knapsack problem (single bin) and Multiple Knapsack Problem (MKP) (Borndorfer and Weismantel, 1997). A detailed survey of sequential algorithms for solving GAP can be found in (Freville, 2004).

Numerous capacity-constrained problems in a wide variety of domains can be abstracted as GAP instances (Cattrysse and Wassenhove, 1992) such as the job-scheduling problem in computer science (Balachandran, 1976), and land use optimization in geographic information science and regional planning (Cromley and Hanink, 1999). Various exact and heuristic algorithms have been developed to solve GAP instances of modest sizes (Freville, 2004). However, in practice, problem instances often have larger sizes while the problem solving requires a rapid solution time and the capability for finding a set of feasible solutions of specified quality, which compounds the computational challenges.

Canonical work on GAP heuristic algorithms include genetic algorithms (Chu and Beasley, 1997; Wilson, 1997; Feltl and Raidl, 2004; Lau and Tsang, 10-12 Nov 1998; Lorena, Narciso and Beasley, 1999), simulated annealing (Qian and Ding, 2007), tabu search (Diaz and Fernandez, 2001), and hybrid search that combines heuristics and local search strategies

(Yagiura, Ibaraki and Glover, 2006; Jeet and Kutanoglu, 2007; French and Wilson, 2007). Previous work applying GAs to GAP indicated that standard GA operations (i.e., selection, crossover, mutation, and replacement) often produce infeasible solutions that violate the capacity constraint of GAP, thus requiring additional processing. For example, Chu and Beasley (1997) developed two additional operators, feasibility improvement and quality improvement, to convert infeasible solutions to feasible ones and attempt to improve each resulted feasible solution, respectively. Wilson's method (Wilson, 1997) allows for infeasible solutions in GA computation, but improves the feasibility of all solutions after GA stops. The method by Feltl and Raidl (2004) also allows for infeasible solutions but with a penalty-based mechanism for adjusting the fitness value of infeasible solutions.

2.3 Parallel Genetic Algorithm Design

The mathematical formulation of GAP is as follows:

$$\text{Objective:} \qquad \min \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{2.1}$$

$$\text{such that:} \qquad \sum_{j=1}^{n} w_{ij} x_{ij} \leqslant b_i, i = 1, 2, ..., m \tag{2.2}$$

$$\sum_{i=1}^{m} x_{ij} = 1, j = 1, 2, ..., n \tag{2.3}$$

$$x_{ij} \in \{0, 1\}, i = 1, 2, ..., m, j = 1, 2, ..., n \tag{2.4}$$

where matrix $C_{m \times n}$ and $W_{m \times n}$ denote the cost and weight requirements of assigning each item $j, j = 1, 2, \ldots,$ to each bin $i, i = 1, 2, \ldots,$ respectively. Constraint (2.2) represents the weight capacity constraint indicating that the total weights of the items assigned to a bin cannot exceed the bin's capacity. $X_{m \times n}$ is the assignment matrix with each entry valued either 0 or 1. Constraint (2.3) is the assignment constraint that allows an item to be assigned to exactly one bin. Each problem instance has the cost and weight matrices as input. The output is the assignment matrix $X_{m \times n}$ that minimizes the cost defined in the objective function.

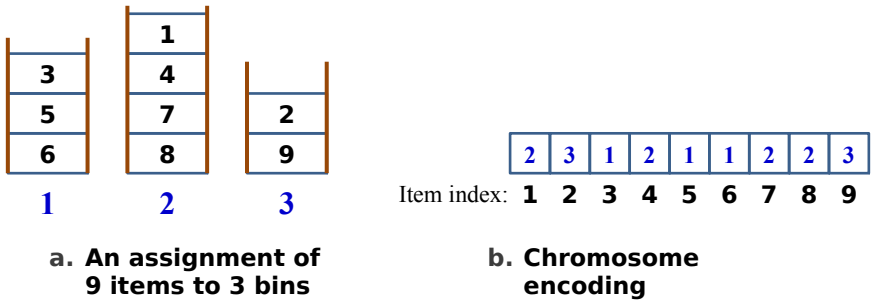


Figure 2.1: GAP encoding.

2.3.1 Sequential Genetic Algorithm

The sequential GA developed for this GAP formulation is an extension of the GA developed by Chu and Beasley (1997). A solution is encoded as a chromosome that is represented as a binary string s of size n, where the value of s_j denotes the index of the bin to which item j is assigned (Figure 2.1). A population is formed by a set of encoded solutions. Population size is a runtime parameter. This GA follows a *steady-state* (Goldberg, 1989) reproduction process in which each iteration selects two parents to generate one child. The GA operators are adapted from Chu and Beasley (1997) and are as follows.

- *Selection*. Two parents are selected based on selection strategies such as binary tournament selection or rank-based tournament selection (Goldberg and Deb, 1991).
- *Crossover*. The simple single cut-point crossover operator randomly decides a cut point at which the first part of a parent is combined with the second part of the other to form the child solution.
- *Mutation*. The mutation operator is performed on the child solution obtained from the crossover operator to exchange the bin assignment of two randomly chosen items.
- *Feasibility improvement*. This operator looks at those over weighted bins after fitness evaluation for a reassignment that could keep the weight sum of the bin below its capacity. This is done by moving an item in an over weighted bin to an under weighted bin.
- *Quality improvement*. This operator looks at each item for a possible reassignment (to another bin) such that the solution's fitness value could be improved.

The fitness of a solution has two components. It has a fitness value that is equivalent to the value of GAP objective function and an "unfitness" value which is the sum of exceeded weights in each bin. For feasible solutions, the unfitness value is always zero. Replacement strategies use the unfitness value to choose the solutions to be replaced. The GA execution stops if the stopping criteria are met. The stopping criteria can be a fixed number of iterations, a time limit, or a given solution quality threshold.

Chu and Beasley (1997) and Feltl and Raidl (2004) indicate that the feasibility and quality improvement operators are critical for keeping the search close to feasible regions in the solution space and helping the GA to converge to near-optimal or optimal solutions quickly. We further refine these two operators to improve lookup efficiency. Both operators include a linear scan of bins to identify either over weighted bins for feasibility improvement or under weighted bins that need to be reassigned an item to improve solution quality. In Chu and Beasley (1997), the order in which bins are checked is fixed, which means that the capacity of each bin is also examined in a fixed order. Considering capacity variations among bins, such examination with the fixed order may limit the search efficiency for possible candidates choices for item reassignment. For example, bins checked first are always considered for reassignment first despite possible alternatives. Such limitations become increasingly significant in PGA as the size of global population is often large. In our algorithm, the bin lookup sequence is determined randomly, resulting in a bin lookup operation that is independent of the order of bins coded in problem instances. The performance impact of this technique is discussed in section 2.4.

2.3.2 PGA Operators

Our coarse-grain PGA, PGAP, employs asynchronous migration for three reasons. First, synchronizing each GA iteration is not necessary in a coarse-grained PGA. Second, the overhead of synchronization gets worse when more cores are used (see section 2.1). Third, non-blocking migration operators increase the overlapping of computation and communication (local evolution can go to the next iteration without waiting for response messages from receiving demes before the next round of migration) and, thus, may significantly improve the computational performance of PGAP.

In PGAP, each deme initializes and maintains a local population. The size of a local population is referred to as deme size. The number of demes is determined based on the number of cores available at runtime. Demes are connected with a regular 2-D toroidal grid topology (Figure 2.2). On the grid, the start and end of a row/column are connected.

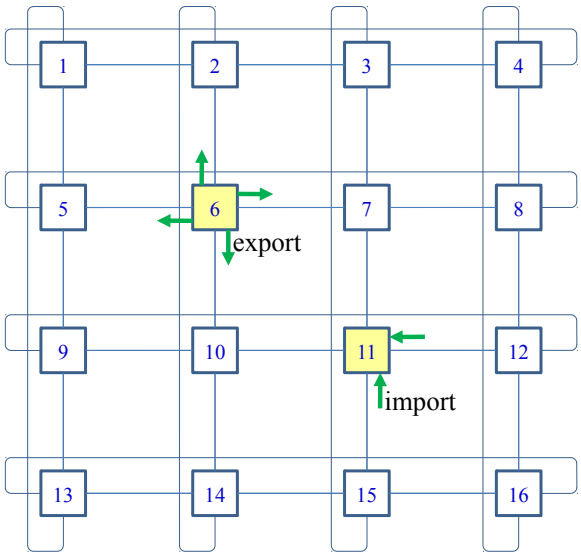


Figure 2.2: Topology of PGAP.

Therefore, the connectivity degree, d, is four for each deme. The algorithm runs concurrently on all cores until a stopping criterion is satisfied at any of the cores.

The following PGA operators are designed to exchange a portion of a deme's population with its directly connected neighbors.

Export. A deme periodically exports a fixed number of solutions to its neighbors. The export operator defines two parameters: migration rate r (i.e., the number of solutions to be exported) and export interval M_{expt} (defined as the number of iterations). Our strategy for exporting solutions considers both elitism and diversifying the deme population by exporting random solutions. When choosing r solutions for exporting, elite solutions generated during the previous export interval are always included. Remaining spots, if any, are filled by randomly picked solutions from local population. If no elite solutions were found in the previous export interval, a holding strategy is applied to probabilistically delay the export.

Import. Each deme maintains an import pool, implemented as a cyclic first-in-first-out (FIFO) queue with the queue header pointing to the first imported solution and queue tail pointing to the next available buffer space, to hold external solutions migrated from neighboring demes. A parameter, import interval M_{impt} (defined as the number of iterations), is used to control how often an import operation is performed. An import operation copies all incoming solutions from the underlying communication system to the import pool. When the pool is full, new incoming solutions override older ones. The size of the import pool is

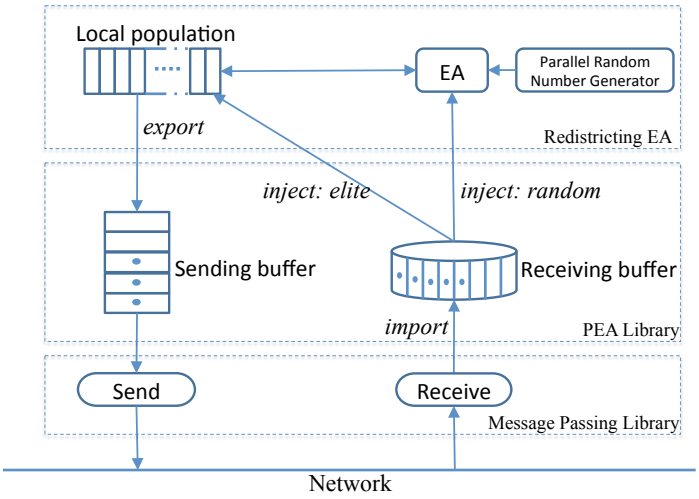


Figure 2.3: PGAP operators and their interactions.

determined by a buffer management method described later (see section 2.3.3) to avoid this overriding scenario.

Inject. The inject operator merges migrated solutions from the import pool into local population. Elite solutions from neighboring demes are alway incorporated into local population if they are superior than local elite solutions. Random solutions from outside are considered as candidates to be selected as one of the two parents for standard GA operations.

Figure 2.3 illustrates these PGA operators and their interactions with the local GA and network. The pseudo code of PGAP is provided in Algorithm 2.1. The introduction of PGA operators affects the selection and replacement operators in the sequential algorithm. After the inject operator is performed, the selection operator takes an injected random solution as one of the two parents for subsequent GA operators. The replacement operator directly merges a remote elite solution into the local population. By doing so, the local population at a deme is able to evolve with better solutions already found by other demes, saving local computation to reach the same level of solution quality.

Furthermore, in addition to the selection strategy in the export operator for sending out local elite solutions, good solutions can be propagated to other demes in a hop-by-hop fashion until they are overtaken by better solutions. This is similar to the diffusion model in fine-grained PGAs (Manderick and Spiessens, 1989). The reason to export random solutions is that these solutions appear as noise to local evolution on the receiving deme and may

influence local evolution paths in a positive way (Hart et al., 1996). When a local evolution process is trapped in local optima or stays within a premature state, such extraneous "noise" may help jump-start the evolution process to search new solution space, thereby providing the diversification effect for local evolution. On the other hand, the probability of holding is kept sufficiently low to avoid excessive interference on any receiving deme's evolution process.

Algorithm 2.1: Parallel genetic algorithm for GAP.

```
1  Individual Pool[bufsize]; // import buffer
2  Generate and evaluate initial population P;
3  iter := 0;
4  do /* evolutionary iteration */
5     c := ∅;
6    select p1 and p2 from P using binary tournament selection; // selection
7    if (there are new solutions in Pool) // migration: inject
8      t := the first unprocessed solution in Pool;
9      if (t is an elite solution from a neighboring deme)
10       c := t; // directly-inject this elite solution into local deme
11     else // a random solution from neighbor
12       p2 := t; // select as a parent
13   if (c is empty)
14     cutpoint := rand(); // randomly select a cutoff point for p1 and p2;
15      c := combine(p1, p2, cutpoint); // crossover
16     /* mutation: swap bin assignment of two randomly selected items */
17     c := swap(c, rand(), rand());
18   get a random lookup order O;
19   foreach bin in O // feasibility improvement on c
20     calculate weight sum for bin in c;
21     if (bin is overweighted)
22       try to move one item from bin to another
23       such that both bins are under capacity;
24   foreach item in c // quality improvement
25     try to move the item to another bin such that
26     the fitness value of c will be less;
27   calculate the fitness value for c; // fitness evaluation
28   if (c is feasible and unique) // replacement
29     if (there exists an individual with unfitness value > 0)
30       replace the unfit individual;
31     else replace the individual with the worst fitness value;
32   if (iter % M_expt = 0) // migration: export
33     if no improvement since last export and should hold
34       skip this export operation;
35     else wait until previous export operation is finished;
36       select and send r solutions to direct neighbors;
37   if (iter % M_impt = 0) // migration: import
38     probe incoming solutions;
39     foreach new incoming solution insoln
40       Pool[++ bufhead ] := insoln;
41   iter++;
42 until stopping rule is met;
```

Compared to synchronous PGA design in which all demes need to stop for communication within the selection operator (in the case of fine-grained PGA) or export and import operators (in the case of coarse-grained PGA), there is no global communication barrier for migration-related communication in PGAP. On each deme, an export operation starts every M_{expt} iterations, returns immediately without waiting for acknowledgement messages from receiving demes, and completes when exported solutions are passed to the underlying communication system. The import operator on each deme checks for incoming solutions every M_{impt} iterations. If no incoming solutions are found, the deme proceeds to the next iteration without waiting. The check operation, called probe, is lightweight. The inject operator injects imported solutions from the import pool into the local evolution process, one at a time. The benefit of using asynchronous migration is two-fold as follows. Globally, asynchronous migration eliminates the costly global coordination among all demes. Locally, the overlap between computation and communication, enabled by the non-blocking export and import operators and buffer-based export communication, increases significantly to allow for better computational performance. Figure 2.2 illustrates a runtime topology of PGAP on which migration events and computation take place at the same time.

The granularity of computation and communication overlap is controlled by allowing a fixed number of sending operations to proceed before pausing future sends until previous sending operations are complete at the network communication level. A parameter, *sending parallelism*, is defined as part of PGA configuration for this purpose. The value of this parameter depends on the underlying network interconnect performance. On a low-latency and fast network, a small value is sufficient to minimize the communication delay on the computation.

2.3.3 Buffer Management for Asynchronous Migration

Buffer-based migration was used before for developing asynchronous PGAs. For example, Andre and Koza (1996) implemented an asynchronous PGA by separating export, import, and other GA operators as independent processes and used per-neighbor buffers to receive exported solutions from corresponding neighbors. To the best of our knowledge, no previous work has considered the issue of buffer overflow and its relationships with PGA parameters. In PGAP, buffer-based communication and non-blocking communication functions are applied together to implement the asynchronous migration strategy and achieve desirable overlapping of computation and communication in export and import. A buffer is used at two places: 1) the import pool is implemented as a cyclic buffer; and 2) PGAP allocates

memory to the underlying communication system as an outgoing message buffer (hereafter, the "sending buffer") for buffer-based non-blocking export operation. Buffer overflow at both places must be considered and avoided whenever possible to make PGAP reliable. The overflow of the import pool will override solutions received in previous import operations, and thus cause the loss of imported neighbor solutions if they have not yet been injected. The overflow of the sending buffer has a more severe impact—the underlying message-passing library often terminates the execution of all processes even when the overflow occurs within one process. We argue that an inappropriate configuration of PGA parameters may lead to serious buffer overflow issues. An algorithmic analysis is conducted on PGAP to study the two buffer overflow issues caused by communication delays. Two algorithmic conditions are derived to detect and avoid buffer overflow issues by setting PGAP parameters appropriately.

PGAP uses message passing (specifically, MPI (MPI-Forum, 2012)) as the underlying parallel programming model and implements the buffer-based non-blocking communication in a manner suggested in (MPI-Forum, 2012):

- Message send operation completes without the need to wait for the post of matching receive at destination process. Instead, the sending operation is considered complete after the message is handed to the sending buffer. Send operation is implemented as non-blocking function;

- The buffered send operation calls a non-blocking non-buffered send for sending each message queued in the buffer, which will not complete until the matching receive is posted or the receiving process starts receiving. This is suggested by MPI standard (see chapter 3 in (MPI-Forum, 2012));

- The sending buffer is managed by the underlying communication system, but the memory of this buffer is allocated explicitly by PGAP;

- Before the message receive operation starts, a probe operation is called to check new messages. Probing introduces negligible cost. If probing does not find new messages, the receive operation is postponed. A receive is considered complete when an incoming solution is copied into the import pool. The receive operation is implemented as blocking function;

- A import operation, if called, receives all of incoming solutions into the import pool.

Table 2.1 lists the PGA parameters considered in our analysis of buffer overflow. Our analysis is based on a common PGA execution environment where each deme has the same

Table 2.1: PGAP parameters considered in the algorithmic analysis.

d	Connectivity: the number of directed connected neighboring demes
r	Migration rate: the number of local solutions selected for each export operation
M_{expt}	Export interval: the number of iterations between two consecutive export operations
M_{impt}	Import interval: the number of iterations between two consecutive import operations
$K_{sendbuf}$	Size of the sending buffer allocated as the outgoing message buffer. $K_{sendbuf} \geq r$
K_{impt}	Size of the import pool

setting for PGA parameters listed on Table 2.1. This analysis focuses on the overflow scenarios caused by communication delays among demes. For simplicity purpose, the holding strategy mentioned in section 2.3 is not considered. Computing time for an iteration is assumed to be consistent across all demes. This assumption indicates that each participating core may have similar CPU and memory characteristics, which is the case for the supercomputers we used for experiments.

2.3.4 Sending Buffer Overflow Analysis

The sending buffer overflows when there are too many pending send operations to handle (Figure 2.3). In PGAP, communication delays in the underlying communication system and/or a long import interval (M_{impt}) on the receiving end can prevent a send operation from being complete. Communication delays are caused by network congestion or communication system resource limitations, which are not able to control in the algorithm design. Allocating a large $K_{sendbuf}$ helps, but does not guarantee the avoidance of overflow. The purpose of our analysis is to avoid the overflow scenarios caused by inappropriate configuration of PGAP parameters.

We consider the worst case scenario in which no send operation can be completed. Suppose it takes x iterations for export operations to fill the sending buffer with $K_{sendbuf}$ solutions. The rate of buffer filling is $\frac{r}{M_{expt}}$ per iteration. By letting $x \times \frac{r}{M_{expt}} = K_{sendbuf}$, we have $x = \frac{K_{sendbuf}}{r} \times M_{expt}$. If M_{impt} at the receiving deme is set to be less or equal to x, we can avoid the sending buffer overflows caused by inappropriate configuration of PGAP. Equivalently, this condition can be written as

$$\frac{M_{impt}}{M_{expt}} \leqslant \frac{K_{sendbuf}}{r} \tag{2.5}$$

where $K_{sendbuf}$ is usually fixed for a PGAP run. Condition (2.5) can then be used to guide the appropriate configurations of M_{impt}, M_{expt}, and r.

2.3.5 Import Pool Overflow Analysis

If the solutions in the import pool of a deme are not injected into the local population in time because of computational noise or outpaced communication between the deme and its neighbors, the pool may overflow. In this overflow situation, one strategy is to allow solution overriding. Such overriding is "harmless" if new solutions are elite solutions from the same neighbor because a newly arrived elite solution always has equal or better quality. However, since we allow random solutions to be exported and elite solutions from different neighbors may have different quality, overriding current solutions in the import pool may result in overriding better solutions that have not been injected into the local population. Sophisticated methods can be developed for import pool management to handle such overriding but at the price of slowing down the processing of the import operator. By looking at overflow scenarios and their relationship to PGAP parameters, a sufficient condition is derived to guide the configuration of K_{impt}, M_{expt}, d, and r in order to avoid the overflow of the import pool.

The import pool will overflow if and only if

1. the rate of producing solutions to the pool is faster than the rate of consuming from the pool, if we consider the import operator as the producer and the inject operator as the consumer; or

2. the import pool is not large enough to hold imported solutions received in a single import operation.

The first condition means if the rate of import operations (in iterations), denoted by R_{impt}, is larger than the rate of inject operations (in iterations), denoted by R_{inject}, the import pool will overflow eventually, regardless of the size of the pool (K_{impt}). Let us denote the event of import pool overflow as O, the number of received solutions in a single import operation as k_{impt}, and the number of solutions remaining in the import pool at the beginning of an iteration as k. Therefore, when an import operation is complete, there are $k + k_{impt}$ solutions in the import pool. From the notations defined in Table 2.1, the above overflow avoidance conditions can be written equivalently as

$$A : R_{impt} \leqslant R_{inject},$$
$$B : k + k_{impt} \leqslant K_{impt},$$
$$A \wedge B \longleftrightarrow \neg O$$

The left part ($A \wedge B$) is the sufficient and necessary condition for avoiding overflow. An exporting deme sends r solutions every M_{expt} iterations to a receiving deme. The receiving rate R_{impt} is the same as sending rate $\frac{d \times r}{M_{expt}}$, otherwise the sending buffer from at least one neighboring deme will eventually overflow. PGAP algorithm (Algorithm 2.1) injects one imported solution per iteration if the import pool is not empty, indicating $R_{inject} = 1$. A is then equivalent to

$$\frac{d \times r}{M_{expt}} \leqslant 1 \tag{2.6}$$

Condition (2.6) indicates that, in each iteration, there is at most one incoming solution sent by neighboring demes. Note that condition (2.6) is implied in B because K_{impt} is a constant. For B, between any two consecutive import operations, neighboring demes constantly sends $d \times \frac{r}{M_{expt}} \times M_{impt}$ solutions to the receiving deme. Communication delays could hold the receiving of these solutions to later import operations. However, since we must avoid the overflow of any sending buffers, the receive operation cannot be postponed infinitely. In fact, by applying condition (2.5) derived in section 2.3.4, we get an upper bound for k_{impt}:

$$k_{impt} \leqslant K_{sendbuf} \times d \tag{2.7}$$

Furthermore, $K_{sendbuf} \times d$ is the largest possible number of solutions in the impool pool. This is simply because $R_{impt} \leqslant R_{inject} = 1$, meaning that the inject operator consumes solutions in the import pool no slower than the import operator producing solutions into the pool. Therefore, delayed receive operations in an import operation is the only possible scenario to make the number of solutions in the import pool following an import operation to be larger than that of the previous import. An import operation with delayed receive operations can receive up to $K_{sendbuf} \times d$ solutions. Meanwhile, since it takes at least $K_{sendbuf} \times d$ iterations to gather these many receiving requests (condition (2.6)), by the time the import operation eventually happens, there will be zero solutions in the pool. Thus, condition $K_{impt} \geqslant K_{sendbuf} \times d$ satisfies B.

Combining (2.6) and (2.7), we get

$$\left(\frac{d \times r}{M_{expt}} \leqslant 1\right) \wedge (K_{impt} \geqslant K_{sendbuf} \times d) \longrightarrow \neg O \tag{2.8}$$

Condition (2.8) shows the correlation among PGAP parameters listed on Table 2.1 in order to avoid the overflow of the import pool. Based on condition (2.5) and (2.8), buffer overflow issues caused by inappropriate configuration of PGAP parameters can be avoided.

2.3.6 Implementation

PGAP is implemented in C. The export and import operators are implemented using the MPI message-passing programming model. MPI's non-blocking point-to-point communication functions *MPI_Ibsend()* and *MPI_Iprobe()* are used for implementing non-blocking features of the export and import, respectively.

The implementation of the outgoing message buffer needs careful investigation. Originally, the sending parallelism is achieved by allocating a sufficiently large user-space memory for MPI to use as the outgoing message buffer, through *MPI_Buffer_attach()*. Since severe communication delays can still cause the overflow of the sending buffer, PGAP implementation skips an export operation if the sending buffer is full. While this solution scaled well on 16K processor cores with fine-tuned PGA parameters based on network performance profiling, it scaled poorly on larger number of faster processor cores on larger scale supercomputers (e.g., BlueWaters) and caused MPI communication layer failure. The outgoing message buffer controlled by MPI experienced buffer overflow as message sending among PGA processes become seriously skewed due to the outpaced runtime delays from numerical operations and non-blocking sending and receiving. The enhanced library manages the sending buffer at the application level and uses *MPI_Isend()* and *MPI_Testall()* to test the completion of previous sending operations in a non-blocking fashion.

The communication topology is generated dynamically by making the sizes of the two dimensions of the 2-D grid as close as possible. The runtime configuration of PGAP parameters is based on the algorithmic analysis results in section 2.3.3.

A random number generator is used to provide stochastic choices on GA operators. Given the fact that GA often requires a large number of iterations before converging to specified solution quality, the same random number sequence cannot be used on all demes even with different initial seeds. A parallel random point generator, SPRNG, is employed to generate a unique random number sequence on each deme (Mascagni and Srinivasan, 2000).

The PGAP solver software is published as an open source and can be accessed online.[1]

[1]Code repository: https://github.com/cybergis/cybergis-toolkit/tree/master/pgap

2.4 Evaluation

As a complex and dynamic process, PGA computation is sensitive to a set of algorithmic and computational factors. Alba and Troya (2002) and Hart et al. (1996) pointed out that conventional parallel computing performance measures, such as speedup and efficiency, need careful consideration when applied to computational performance evaluation of PGA. Since PGA and GA are algorithmically different, direct performance comparison between PGA and sequential GA may not lead to appropriate conclusions. Alba and Troya (2002) further suggest using the same parallel version on different number of cores for speedup measurement. Also, a sufficient number of trials should be conducted to yield statistically confident evidence on obtained results (Hart et al., 1996; Whitley, 2001). Identifying the sources of performance variation in asynchronous PGA is another issue. Gordon and Whitley (1993) observed that even on a single core, emulated PGAs were often more efficient than many sequential GAs for solving various types of problems. In addition, injected computational noise changes asynchronous PGA algorithmic performance (Hart et al., 1996), making performance measurement more complicated.

In high-performance computing, weak and strong scaling are two typical ways to measure the scalability of high-performance computing algorithms (Clausen, Reasor and Aidun, 2010; Dongarra et al., 2007). For weak scaling, problem size per core is kept constant as the number of cores increases. Weak scaling thus allows us to look at the capability of an algorithm to solve larger or more complicated problems in conjunction with the use of more cores. Strong scaling keeps the overall problem size constant as the number of cores varies, and measures speedup, calculated by dividing the execution time of the best sequential algorithm by that of the parallel algorithm. However, neither of them can be directly applied to analyze the scalability of PGAs for solving combinatorial optimization problems (elaborated in 2.4.2). This research, therefore, has designed specific scaling test methods for the comprehensive evaluation of the performance of PGAP.

2.4.1 Experiment Design

Experiments using up to 131,072 cores were designed to evaluate the scalability of the PGAP algorithm. Both strong and weak scaling tests, tailored for PGA performance evaluation, were performed. Strong scaling tests were conducted to measure speedup in a set of large-scale PGAP runs, each using a different number of cores. However, two issues related to PGA performance evaluation, defining base case runs and comparing speedups in runs that achieve different solution quality, need to be resolved. In weak scaling tests, population

size, instead of problem size, is used as scaling factor in order to appropriately measure the computational effort required in relation to the increase of computing power. For both tests, results were compared with the corresponding synchronous implementation. PGA algorithmic capabilities such as numerical performance, solution quality improvement, and convergence were evaluated to understand the improved problem-solving capabilities by using the asynchronous migration strategy.

Three MIMD high-performance computing (HPC) systems were used for the experiments: Ranger and Stampede at the Texas Advanced Computing Center (TACC) and Blue Waters at the National Center for Supercomputing Applications (NCSA). Ranger has 62,976 cores with 0.579 petaflop peak performance. Each node on Ranger contains four 2.3GHz AMD Opteron quad-core processors (16 cores) and 32GB memory. InfiniBand is the interconnect. Up to 16,384 cores on Ranger were used for scalability tests and numerical performance evaluation in solving large GAP instances, as well as for studying the convergence of PGAP and tracking the number of feasible solutions found in PGAP runs.

Stampede, a newer and larger supercomputer, is a 10 petaflop system with more than 6400 computing nodes, each is equipped with 16 cores on the Intel Xeon E5 Sandy Bridge processor (2.7GHz). A portion of the system has the Intel Knights Corner (KNC) coprocessor (MIC architecture; 1.4GHz; 61 cores/node). The interconnect on Stampede is a new generation InfiniBand with lower latency. Due to the heterogeneous configuration of the host and coprocessor, Stampede was used to test our library's resilience to computing and communication variations. On both Ranger and Stampede, MVAPICH2 is used as the MPI library for optimized performance on InfiniBand interconnect.

Blue Waters, the most powerful supercomputer in an academic computing cyberinfrastructure, is an 11 petaflop system with the AMD Bulldozer processor (2.3 GHz, 32 integer cores and 16 floating point cores per node). It employs Cray's Gemini interconnect, configured as a 3D torus. The interconnect has extremely low latency. Up to 131,072 cores on Blue Waters was used to test the our library's performance at an extreme scale. Cray MPI was used to compile the library.

Five types of public GAP benchmark instances are available from OR-LIB[2] and Yagiura's website[3] have been used in the literature as benchmark datasets. Small-sized type D, E, and F instances from OR-LIB are used in small-scale experiments to verify the quality of solutions found by our baseline GA algorithm. Type D and E instances from Yagiura are large instances and considered more difficult because costs are inversely correlated with

[2]http://people.brunel.ac.uk/~mastjjb/jeb/info.html
[3]http://www-or.amp.i.kyoto-u.ac.jp/~yagiura/gap/

Table 2.2: PGAP default configuration.

Parameters	Settings
Population size per deme	100
Initial population generation	Random with feasibility improvement or constraint-based improvement (Feltl and Raidl, 2004)
Selection	Binary tournament
Crossover	1-point. Probability: 0.8
Mutation	1-item mutation. Probability: 0.2
Replacement	Replacing the unfittest or worst
Elitism	Yes
Stopping rules	No solution improvement, bounded solution quality reached, or fixed number of iterations
Process topology	2-D Torus
Number of islands per processor	1
Process connectivity d	4 (number of direct neighbors of each process)
Migration rate r	2 (number of solutions sent to neighbors in each export)
Export interval M_{expt}	50 (number of iterations between two consecutive exports)
Import interval M_{impt}	25 (number of iterations between two consecutive imports)
Sending parallelism p	2 (number of exports to invoke before waiting for their finish)
Sending buffer size $K_{sendbuf}$	4 solutions. Actual memory requirement is $(p \times d \times n \times 4 + \text{buffer\_overhead})$ bytes
Import pool size K_{impt}	80 solutions. Actual memory requirement is $(80 \times n \times 4)$

weights (Amini and Racer, 1994). E801600, one of the largest instances of type E with 1,600 items and 80 bins, was used for PGAP scaling tests and performance evaluation.

Table 2.2 shows the default configuration of PGAP parameters based on condition (2.5) and (2.8) derived in section 2.3.3. The ratio M_{impt}/M_{expt} is configured to be much smaller than $K_{sendbuf}/r$ in order to reduce the impact of sporadic network congestions on the sending buffer. We also follow the guidelines by Hart et al. (1996) and Alba and Troya (2002) to make sure that results from different PGAP runs are comparable. For example, the stopping rule for speedup analysis is finding the best solution found by the base case runs, instead of setting a walltime or the number of iterations. A synchronous PGA is developed based on PGAP as the reference implementation for performance comparison purpose. This is done by adding global barriers for export and import operators to PGAP and using blocking communication functions and synchronous communication mode in MPI. Therefore, synchronization cost is the only source of performance difference.

Results from the experiments are presented in the following sections. The results on scalability (section 2.4.2), solution quality (section 2.4.4), and numerical performance (section 2.4.5) were obtained on Ranger. In section 2.4.7, similar tests were run on Stampede to test

the the difference under faster computing nodes and network with lower latency. Results using both host and KNC coprocessors are reported in section 2.4.7. Section 2.4.9 reports the scalability of the PGAP library using the unprecedented numbers of processor cores, up to 131,072.

2.4.2 Scalability Analysis

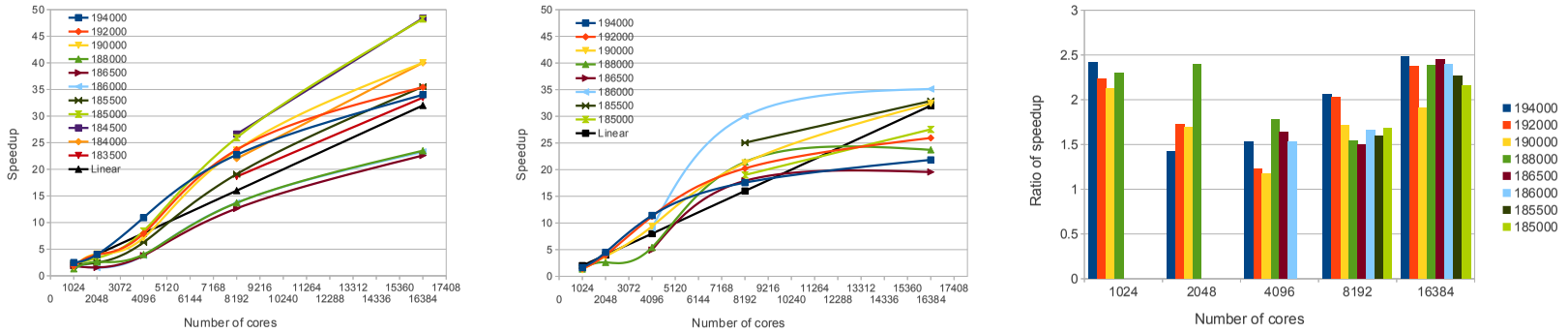
It is not appropriate to directly use conventional weak and strong scaling testing methods to analyze the scalability of PGAP for the following reasons. First, the amount of computation work required to solve a combinatorial optimization problem depends as much on the problem difficulty as on the problem size. For example, a small-size (in terms of the number of items and/or bins) type E GAP instance may be "harder" than a large-size type C instance because type E instances were generated by inversely correlate the cost and weight of each item (Yagiura, Ibaraki and Glover, 2006). Therefore, varying problem size only as done in typical weak scaling experimentation may not be sufficient to reveal how well PGA can solve large problems using more computational resources. Second, since PGA is a type of randomized algorithm, multiple runs of the same PGA configuration may require different execution times to achieve the same level of solution quality. Furthermore, while the overall problem size is fixed in strong scaling, changing the number of cores also changes deme size and the number of demes handled by each core. Such changes have a profound impact on local evolution and global migration effect, together complicating the overall evaluation of computational performance.

Our strong scaling experiment measures the speedup of PGAP by increasing the number of cores, but keeping the size of global population constant. Each core runs one deme and thus deme size, which is equal to the global population size divided by the number of cores, varies accordingly. Speedup is calculated as the ratio of the execution time of two PGAP runs with the denominator being a reference (referred to as base case) using a small number of cores. The reason to use PGAP itself as the reference algorithm is that PGA and GA are not suitable for direct comparison since they have different algorithmic behaviors. This definition of speedup is referred to as *relative speedup* by Sun and Ni (1990) or *type I speedup* by Alba and Troya (2002). Since the global population size is kept constant, the increase of the number of cores leads to the decrease of deme size and the increase of the number of demes. It is known that PGA performance is highly influenced by deme size and migration. Specifically, a large deme size makes GA search on a core tend toward a random search and injects solutions from fewer demes, while a small deme size makes local GA search

"premature" but injects solutions from more demes. Our strong scaling test is designed to compare the speedup between PGAP and its synchronous implementation.

In the experiment, global population size was kept at 1,638,400. As the number of cores used in the experiment increased from 512 to 16,384, population size on each deme decreased from 3,200 to 100. Since comparing with sequential algorithm performance would produce misleading results, we used a 512-core run of PGAP as the base case for PGAP. A 512-core run of its synchronous version is then used as the base case for speedup calculation for the synchronous version. Therefore, perfect linear speedup in our experiment would be 32 when 16,384 cores are used. When a specified solution quality threshold could not be reached within the maximum walltime allowed using 512 cores, the execution time of the base case run is set to be the maximum walltime (i.e., 16 hours (57,600 seconds)). In such scenario, using the maximum walltime in the base case is a conservative estimation of the actual speedup in a speedup comparison because the resulted speedup is the lower bound of the actual speedup. Figure 2.4(a) and Figure 2.4(b) illustrate the speedup measurements against different solution quality thresholds, quantified as the upper bound of fitness value. Results show that, for both synchronous and asynchronous runs, using more cores resulted in better speedup, even superlinear speedups. By looking at speedups achieved at multiple solution quality thresholds, a more comprehensive view of PGAP's numerical performance was obtained by examining the relationship between the levels of difficulty to reach a specified solution quality and the number of cores used. For example, for the type E instance, both synchronous and asynchronous runs experienced difficulty trying to achieve a solution quality threshold 188,000, indicated by the sublinear speedup achieved. Asynchronous PGAP runs (Figure 2.4(a)) exhibited superlinear speedup at 8 out of 11 solution quality thresholds when using 16,384 cores, while 3 out of 8 were observed in synchronous runs (Figure 2.4(b)). Synchronous runs could not reach the three tightest solution quality thresholds reached by PGAP, while asynchronous runs were also able to find solutions better than the tightest threshold, 183,500, when using 8,192 and 16,384 cores. Beyond 8,192 cores, speedup increase in the synchronous runs became insignificant, while PGAP scales well to 16,384 cores.

Note that Figure 2.4(a) and Figure 2.4(b) cannot be directly used to compare speedups between PGAP and its synchronous version. This is because they used different base cases for speedup calculation (512-core runs of PGAP and its synchronous version, respectively). Instead, we define *ratio of speedup* as below for comparing speedup differences between asynchronous and synchronous migration strategies. It is measured as the ratio of execution



(a) Asynchronous migration

(b) Synchronous migration

(c) Ratio of speedup comparison

Figure 2.4: Strong scaling results. Speedups were measured against different solution quality thresholds. Missing points/-columns were the runs exceeding maximum walltime.

time of the synchronous version and that of PGAP:

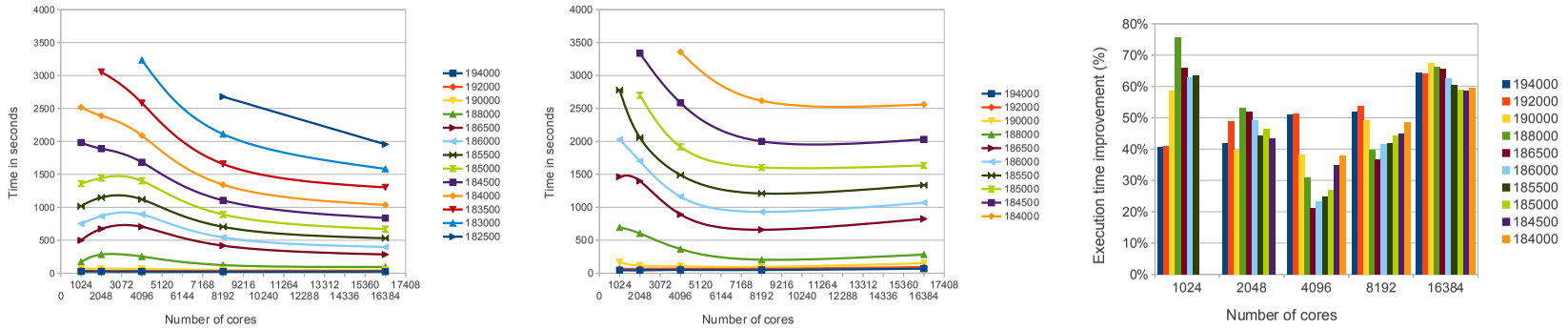
$$ratio\_of\_speedup = \frac{speedup_{async}}{speedup_{sync}} = \frac{T_{base}/T_{async}}{T_{base}/T_{sync}} = \frac{T_{sync}}{T_{async}} \tag{2.9}$$

where T_{base} is the execution time of a common base case. Figure 2.4(c) shows the ratio of speedup, calculated based on equation 2.9 for all of the solution quality thresholds that the synchronous version achieved in at least one setting of the given numbers of cores. For all measurable cases, the values of the ratio are larger than 1, meaning that PGAP achieved better speedup than its synchronous version. Particularly, when using 16,384 cores, the ratio ranges from 1.90 to 2.47.

Because increasing the problem size may alter problem difficulty and the amount of computation needed to solve the problem, problem size, as used in typical weak scaling test, is not appropriate to use in weak scaling test of PGAs. Instead, our weak scaling test varies the size of global population in order to measure how PGAP leverages larger global population numerically, enabled by the use of more cores, to diversify the search in solution space and achieve a designated solution quality more effectively. We refer to this weak scaling test method as *population scaling*. The benefit of using a large population is straightforward because it provides a much larger sampling solution space for GA. But using a large population can easily turn a GA evolutionary process into a random search that may have difficulty converging even on a single core. With PGA, however, the evolutionary process can be kept effective at the deme level, but much more solution space can be searched by running a large number of demes simultaneously.

It is worth noting that population scaling adds more cores to do additional work on the same problem, similar to the "new-era" weak scaling proposed by Sarkar, Harrod and Snavely (2009). Leveraging larger global population, population scaling is designed to evaluate the capability of PGAP to obtain better solutions in a shorter amount of time. The test was done on the Ranger supercomputer. We use the time taken to achieve a certain solution quality as the measure of population scaling. Deme size was set to 200. The number of cores used ranges from 1,024 to 16,384. As the number of cores doubles, global population size also doubles from 204,800 to 3,287,400. We ran this test on both PGAP and its synchronous version. Each run was given one hour to finish.

The benefit of using large population PGAP can be shown in Figure 2.5(a). The time taken to achieve a certain solution quality threshold, quantified again as the upper bound of fitness value, was measured in accordance with the increase in the number of cores. Overall,



(a) Aynchronous migration (b) Synchronous migration (c) Execution time improvement

Figure 2.5: Weak scaling results. Execution times were measured against different solution quality thresholds. Missing points were the runs exceeding maximum walltime (one hour).

as the bound became tighter, it took more time to find solutions whose fitness values are equal to or better than the bound. For a particular bound, it is obvious that using more cores reduces the time taken to find solutions with equal or better quality. This trend became more significant for tighter bounds. In fact, for those bounds tighter than 184,000, runs using smaller number of cores started running out of time to find solutions of specified quality. For example, solutions with bound 182,500 or better were only found by using 8,192 and 16,384 cores. For all of the runs, using 16,384 cores significantly reduced the time taken to achieve the specified solution quality. This can be explained. First, running massive demes simultaneously can greatly increase the overall probability of finding new elite solutions. Second, migration operators are able to propagate good solutions to all demes if they are globally better, stimulating local evolutionary processes at deme level. There were some variations observed. For the test problem instance, improving solution quality from threshold 188,000 to 186,500 was difficult with the given PGA configuration in both asynchronous and synchronous versions. In the asynchronous version, higher fluctuations were observed in one of the runs of using 2,048 and 4,096 cores, respectively, and had subsequent effect on reaching tighter bounds. Such variation was due to the dynamics in stochastic computation specific to each run.

In contrast, Figure 2.5(b) shows the weak scaling performance of the synchronous version. Consistently, not only the time taken to reach a bound was longer than PGAP, the synchronous runs could not find solutions better than 184,000 given the same amount of execution time (1 hour). Figure 2.5(c) shows this trend more clearly by comparing the execution time improvement, measured as the percentage of $\frac{execution\_time_{sync} - execution\_time_{async}}{execution\_time_{sync}}$. When using 16,384 cores, the execution time improvement was consistently around 60%. In summary, the weak scaling experiment shows that asynchronous migration was able to exploit large population size more effectively and showed significant numerical performance advantages over the corresponding synchronous version as more cores were used.

2.4.3 Communication to Computation Ratio

One advantage of using asynchronous migration in PGA is improved communication to computation ratio. Such advantage becomes more obvious when using a large number of cores. In any synchronous PGA implementation, a delay at one core could have significant reverberating effects, while such delays in asynchronous migration only affect direct neighbors (in receiving solutions) and can even be offset by the overlapping of computation and communication.

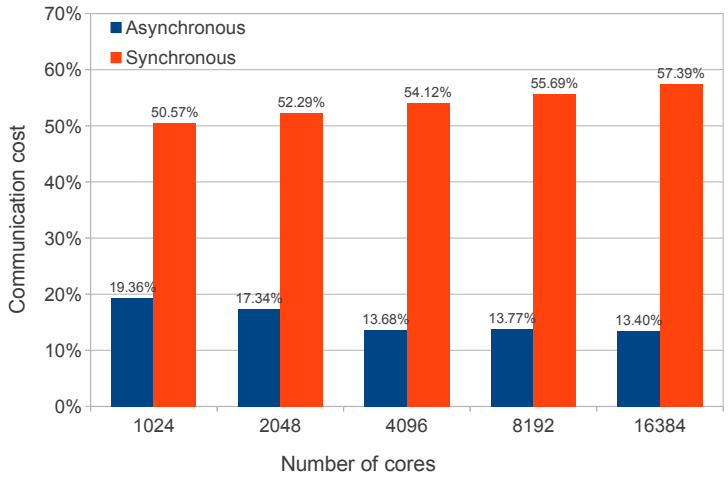


Figure 2.6: Distribution of communication cost in weak scaling test.

In this experiment, the E801600 problem instance was solved by both asynchronous and synchronous versions of PGAP with the same settings as specified in the weak scaling experiment. Figure 2.6 shows the communication cost as percentage of total execution time. PGAP's communication cost (on average 15.5%) was significantly lower than the synchronous version (on average 54%). For the synchronous version, the communication cost increased steadily as the number of cores doubled, mainly due to the increase cost of *MPI_Barrier()* calls. While for PGAP, the communication cost decreased as more cores were used. The decrease can be explained as follows. In a GA execution, as the evolutionary process continues, it becomes harder to find better solutions. Therefore, random solutions are more likely to be selected and migrated. A holding strategy is applied in both PGAP and its corresponding synchronous version to delay the export operation in this situation in order to avoid excessive injection of randomness. As more cores are used, PGAP achieves a solution bound earlier, beyond which getting better solutions takes longer time and the holding strategy is applied more frequently. This scenario applies to the synchronous version, too. But the expensive *MPI_Barrier()* cost exhibited at large scales is more significant than the reduction of communication caused by the holding operations. This observation indicates that optimal search strategies should be employed to keep searching efficiency on pace with the increase of computing power.

To better understand the communication variation in asynchronous migration, a snapshot of communication cost on each core is plotted for a run using 16,384 cores, shown in Figure 2.7. Most of cores' communication cost was around 13%, with a few cores ranging between

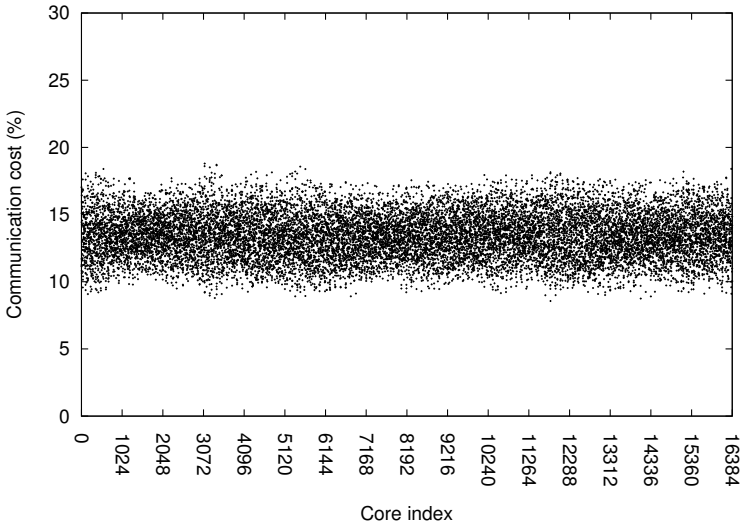


Figure 2.7: Snapshot of communication cost at each core in a PGAP run using 16,384 cores.

8.5% and 18%. This means that communication cost in this large-scale PGAP run was consistent across cores despite that migration operations were not synchronized. Also, the asynchronous migration strategy was able to manage the 9.5% communication cost variation. This experiment further illustrated that asynchronous PGA, if designed properly, can also be reliable while introducing significantly lower communication cost than synchronous PGAs.

2.4.4 Solution Quality of Baseline GA

This experiment was designed to evaluate solution quality of the baseline sequential GA of PGAP, which enhanced the feasibility and quality improvement operators used in Chu and Beasley (1997). As mentioned in section 2.3, randomness was introduced in the feasibility and solution quality improvement operators as a way to diversify search patterns for item reassignment as PGAP scales to use large amount of demes. This algorithmic change improves PGA performance in both spatial and temporal contexts. For all of the demes spatially distributed in PGA computation, the improved operators enable more flexible search patterns on all demes with additional help from using a unique random number sequence per deme. Temporally, they allow each iteration to use a different search order from previous iterations for finding alternative item reassignments. Therefore, the change we made to introduce more randomness also improves the baseline sequential GA. In this experiment, GA parameters were set to mimic those in (Chu and Beasley, 1997), i.e., population size was 100 and the stopping rule was without improvement in consecutive 500,000 iterations. We ran the base-

line algorithm and the two-deme PGAP cases only once. Results were compared with the best-so-far GA results (Feltl and Raidl, 2004) to the authors' knowledge. Table 2.3 lists the best solutions found and compares the gap to the maximum lower bounds (found by the linear programming package (IP/LP Copt)) found among CPLEX commercial software (IBM, 2013), CRH-GA algorithm (Feltl and Raidl, 2004), and our baseline GA. Experiment results showed that our baseline GA outperforms Feltl's GA in 31 out of 39 small-scale type D, E, and F instances. The solution quality improvement is significant because results reported in (Feltl and Raidl, 2004) show the best among multiple runs.

Table 2.3 also illustrates the immediate benefit of using two demes with migration on a single core (column "Two-deme PGAP"). In the two-deme case, the single core take turns to run two demes with migration. Each deme is assigned with a different random number sequence that generates different solution search paths. The benefit of using multiple demes with migration is obvious for the GAP problem. Even though the two demes ran on a single core, Table 2.3 shows that it effectively improved solution quality and outperformed Feltl's GA in 37 instances, and our sequential GA in 31 instances.

2.4.5 Numerical Performance

The impact of asynchronous migration on the numerical performance of PGA is twofold. First, its reduces communication cost allows PGA to run more iterations within the same amount of time. Second, migration intervals become more dynamic among spatially distributed demes. In the case of synchronous migration, a migration operation is followed by a *silent* time among all demes, the length of which is determined by the parameters of migration interval. But in the case of asynchronous migration, as computation goes on, the timing for migration shifts gradually among demes because of runtime dynamics such as system clock and communication delays. Hart et al. (1996) demonstrated the benefits of such dynamics on improving PGA performance in terms of the total number of function evaluations required to achieve a designated solution quality. Alba, Luque and Troya (2004) demonstrated the effect of network latency of LAN (local area network) and WAN (wide area network) on PGA efficiency when they found that WAN executions could be more efficient due to longer isolation times. We used the result of the weak scaling experiment to evaluate numerical performance of PGAP.

To understand the benefit of reduced communication cost, we use the number of iterations performed per second as a measure (referred to as *iteration rate*) to evaluate PGAP's numerical efficiency. In a multi-deme case, iteration rate is measured as the average value

Table 2.3: Solution quality comparison [4][5][6][7] among CPLEX software, CRH-GA, the baseline sequential GA, and 2-deme PGAP.

Prob.	Size		IP/LP	CPLEX	CRH-GA	Baseline GA		Two-deme PGAP	
Type	m	n	Copt	gap (%)	gap (%)	fitness value	gap (%)	fitness value	gap (%)
D	5	400	25670	opt	**0.44**	25790	**0.47**	25737	**0.26**
D	10	400	25274.8	0.18	**1.31**	25509	**0.93**	25435	**0.63**
D	20	400	24546.8	0.51	**1.97**	24903	**1.45**	24855	**1.26**
D	40	100	6092	3.96	**3.72**	6327	**3.86**	6265	**2.84**
D	40	200	12244.9	2.22	**3.06**	12561	**2.58**	12520	**2.25**
D	40	400	24371.8	1.1	**2.81**	24851	**1.97**	24801	**1.76**
D	80	100	6110.5	6.6	**7.01**	6526	**6.80**	6501	**6.39**
D	80	200	12132.3	2.87	**3.76**	12490	**2.95**	12454	**2.65**
D	80	400	24177	2	**3.02**	24748	**2.36**	24587	**1.70**
E	5	100	7757	opt	**0.24**	7774	**0.22**	7760	**0.04**
E	5	200	15611	opt	**0.23**	15632	**0.13**	15626	**0.10**
E	5	400	30794	opt	**0.28**	30872	**0.25**	30826	**0.10**
E	10	100	7387.8	0.61	**0.91**	7454	**0.90**	7436	**0.65**
E	10	200	15039.8	0.25	**0.96**	15135	**0.63**	15126	**0.57**
E	10	400	29977.9	0.09	**0.94**	30135	**0.52**	30142	**0.55**
E	20	100	7348.2	1.32	**1.74**	7463	**1.56**	7448	**1.36**
E	20	200	14765.2	0.89	**1.7**	14923	**1.07**	14918	**1.03**
E	20	400	29500.3	0.34	**1.6**	29845	**1.17**	29810	**1.05**
E	40	100	7316.1	3.32	**3.11**	7497	**2.47**	7522	**2.81**
E	40	200	14630.4	1.85	**2.2**	14835	**1.40**	14858	**1.56**
E	40	400	29186.6	0.69	**2.14**	29586	**1.37**	29593	**1.39**
E	80	100	7650	opt	**0.78**	7670	**0.26**	7668	**0.24**
E	80	200	14566.7	2.17	**2.93**	14833	**1.83**	14846	**1.92**
E	80	400	29161.3	1.57	**2.49**	29631	**1.61**	29567	**1.39**
F	5	100	2755	opt	**0.41**	2761	**0.22**	2761	**0.22**
F	5	200	5294	opt	**0.35**	5304	**0.19**	5315	**0.40**
F	5	400	10745	opt	**0.25**	10776	**0.29**	10765	**0.19**
F	10	100	2276.8	1.99	**3.95**	2364	**3.83**	2348	**3.13**
F	10	200	4644.6	1.13	**3.12**	4794	**3.22**	4759	**2.46**
F	10	400	9372.7	0.46	**2.78**	9631	**2.76**	9604	**2.47**
F	20	100	2145.1	8.15	**8.38**	2339	**9.04**	2301	**7.27**
F	20	200	4310.1	4.73	**6.55**	4585	**6.38**	4552	**5.61**
F	20	400	8479.4	2.38	**7.15**	9050	**6.73**	8950	**5.55**
F	40	100	2110.1	21.28	**18.27**	2476	**17.34**	2501	**18.53**
F	40	200	4086.5	10.14	**12.86**	4522	**10.66**	4578	**12.03**
F	40	400	8274.3	4.05	**10.36**	9105	**10.04**	8907	**7.65**
F	80	100	2064.4	31.37	**26.77**	2655	**28.61**	2583	**25.12**
F	80	200	4123.4	19.05	**17.33**	4869	**18.08**	4837	**17.31**
F	80	400	8167.1	9.18	**12.55**	9248	**13.23**	9127	**11.75**

[4]Column IP/LP: lower bound found by linear programming

[5]Column CPLEX: best solutions found by CPLEX. *opt*: optimal solution found

[6]gap (%): the percentage of $\frac{fitness\_value_{best\_solution} - lower\_bound}{lower\_bound}$

[7]IP/LP, CPLEX, and CRH-GA results are from Feltl *et al.* (Feltl and Raidl, 2004). They are included for comparison purpose

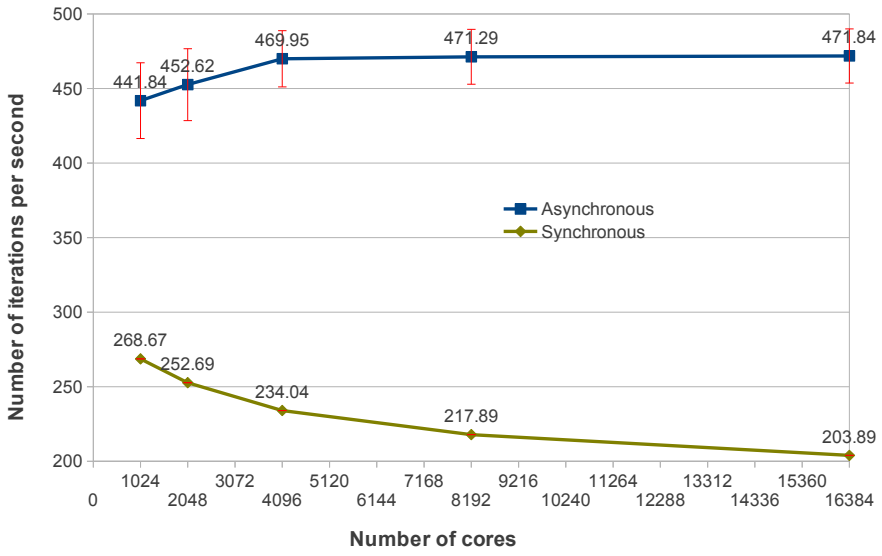


Figure 2.8: Iteration rate comparison. The standard deviation is also plotted.

across all demes. Figure 2.8 shows the result of using 1,024–16,384 cores. The result on the synchronous version is easy to understand. The decrease of iteration rate is due to the increasing cost of *MPI_Barrier()* as more cores are involved. Additionally, because of the use of global barrier, the standard deviation is marginal. Compared to the synchronous case, the benefit of using asynchronous migration is significant, indicated by much higher iteration rate. We also observed that, as more cores were used, the iteration rate did not decrease. Instead, the rate increased from 441.84 to 471.84. Such increase was due to more frequent invocations of the holding strategy. The standard deviation for the asynchronous case decreased from 25.39 to 18.14 as the number of cores increases from 1,024 to 16,384. This may not suggest to use more cores in order to get more stable iteration rate. Instead, the decreasing standard deviation was, again, due to more frequent holding operations caused by earlier PGA convergence in asynchronous runs. In summary, the iteration rate analysis clearly shows that the asynchronous migration strategy in PGAP is highly scalable to the number of cores and the consistent iteration rate allows PGAP to finish more evolution iterations collectively by using more computing power.

By performing more iterations on a problem instance, the improvement of solution quality is expected for PGAP. Figure 2.9 shows how much PGAP can outperform the synchronous version in obtaining better solutions within one hour of computation. We measured solution quality gain over the synchronous version as the percentage of $\frac{fitness\_value_{sync} - fitness\_value_{async}}{fitness\_value_{sync}}$.

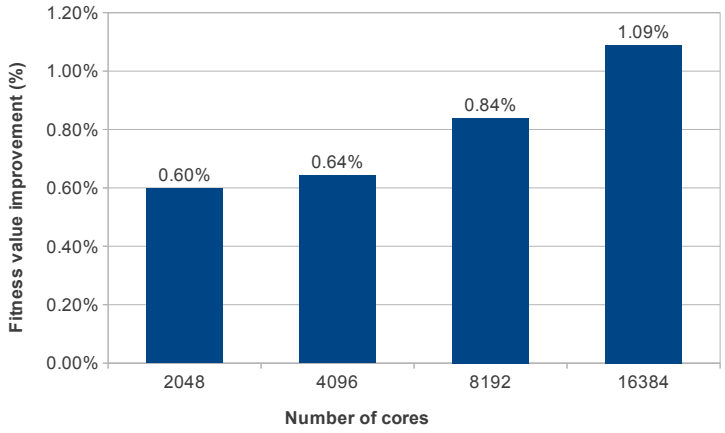
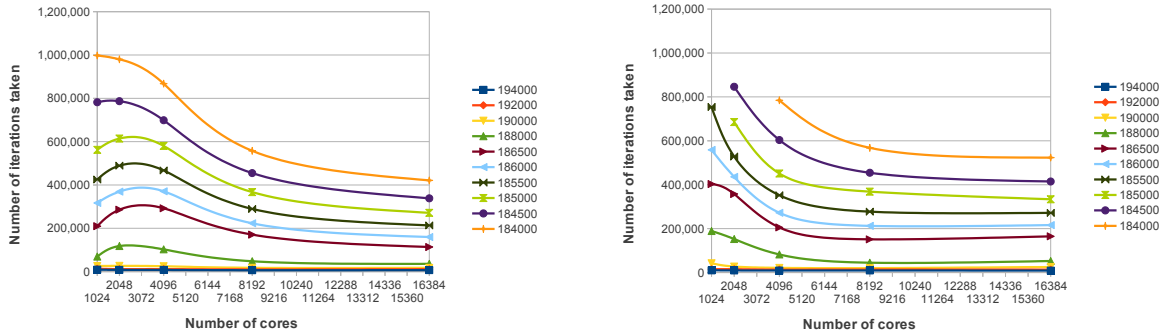


Figure 2.9: Solution quality comparison, measured as the percentage of fitness value improvement in PGAP over the synchronous version. Each run was given one hour to finish.

For all of the cases we evaluated, solution quality of the asynchronous version is better than the synchronous one. The largest improvement of 1.08% occurred when using 16,384 cores, which improved fitness value by 1,999. The percentage of improvement increases as the number of cores doubled.

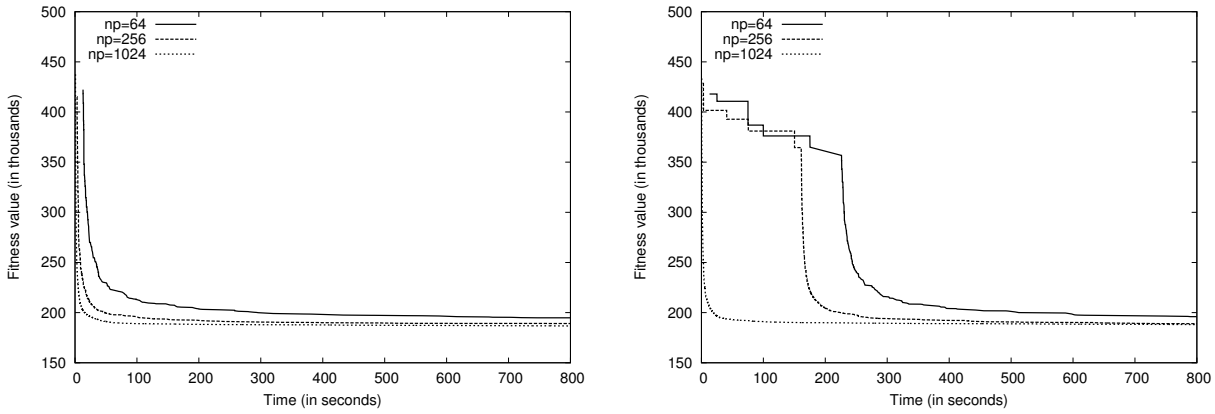
While the improvement of numerical performance from reduced communication cost is apparent, the influence of the migration strategy itself on PGAP's problem-solving capability can be intricate due to the high variation on the timing of migration operations among a large number of demes. A preliminary study was conducted to measure such runtime influence in PGAP by calculating the number of iterations needed to achieve a set of solution quality thresholds on Ranger, instead of measuring the execution time which is highly correlated with communication cost. Figure 2.10 shows the result obtained from the weak scaling experiment. The number of iterations needed in each run is calculated as the minimum number of iterations taken among all of the demes that reached the solution bound. For the cases of 1,024, 2,048, and 16,384 cores, PGAP took fewer iterations to reach any of the ten solution quality thresholds. But for the cases of 4,096 and 8,192 cores, more iterations were needed than the synchronous version to improve from threshold 188,000 to 186,500. Combined with figure 2.5, the numerical performance gain from PGAP seems to mainly attribute to the significant reduction of communication cost. But for the case of 16,384 cores, we consistently observed better performance of PGAP in both execution time and the number of iterations in achieving a specified solution quality.



(a) Aynchronous migration

(b) Synchronous migration

Figure 2.10: Number of iterations required to achieve specified solution quality. Missing points were the runs exceeding maximum walltime (one hour).



(a) Asynchronous migration

(b) Synchronous migration

Figure 2.11: PGAP convergence. np: number of cores.

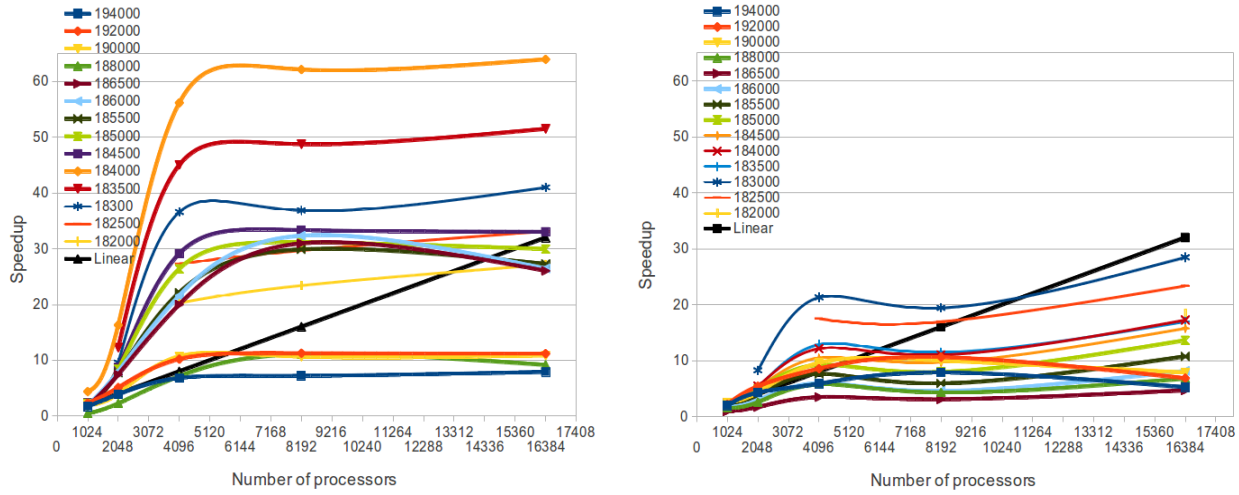
The advantage of using asynchronous migration is also illustrated by the convergence experiment. Figure 2.11 depicts snapshots of the fitness value of the best solutions found at each timestamp in a set of runs of PGAP (Figure 2.11(a)) and its synchronous version (Figure 2.11(b)) using 64, 256, and 1,024 cores on the Lonestar supercomputer. PGAP converged much faster in all cases.

2.4.6 Finding Feasible Solutions

The ability to identify more feasible solutions by leveraging massive computing resources is an important factor for illustrating how large-scale PGAP runs explore solution space more efficiently than its synchronous version. Obtaining more feasible solutions of specified solution quality is often one of the problem-solving goals for a lot of GAP applications. By investigating the landscape of found feasible solutions, better problem-specific heuristics can be developed to guide solution space search toward promising directions while avoiding being trapped in local optima. The capability of PGAP for finding feasible solutions is measured by counting the number of unique solutions found with equal or better solution quality than the specified quality thresholds within one hour of weak-scaling test runs. PGAP runs using 16,384 cores were compared with the runs of its synchronous version. Results showed that, on average, PGAP found 11,093 unique solutions with fitness value better than 194,000, 19.98% more than the synchronous version. This trend became more obvious as solution quality thresholds became tighter. At thresholds of 188,000 and 186,000, PGAP found 40.62% and 75.90% more unique solutions than the synchronous version, respectively. Such improvement in problem-solving capability is mainly attributed to the fact that, given the same amount of execution time, the asynchronous migration strategy allows PGAP to perform significantly more iterations.

2.4.7 Results on Stampede

Figure 2.12 shows the speedup results on Stampede. Similar to the results on Ranger, asynchronous runs outperformed synchronous runs and achieved significantly better speedup. With the asynchronous migration strategy (Figure 2.12(a)), more superlinear speedups were achieved on Stampede, compared to the results on Ranger (Figure 2.4(a)). Such improvement attributes to reduced communication cost on a better InfiniBand interconnect on Stampede, shown in Figure 2.13. The communication cost using 16,384 processor cores dropped from 13.40% on Ranger to 0.06%. The synchronous runs, however, obtained slower speedups (Figure 2.12(b)) than on Ranger (Figure 2.4(b)). This occurs because, first, the synchro-



(a) Aynchronous migration

(b) Synchronous migration

Figure 2.12: Speedups on Stampede.

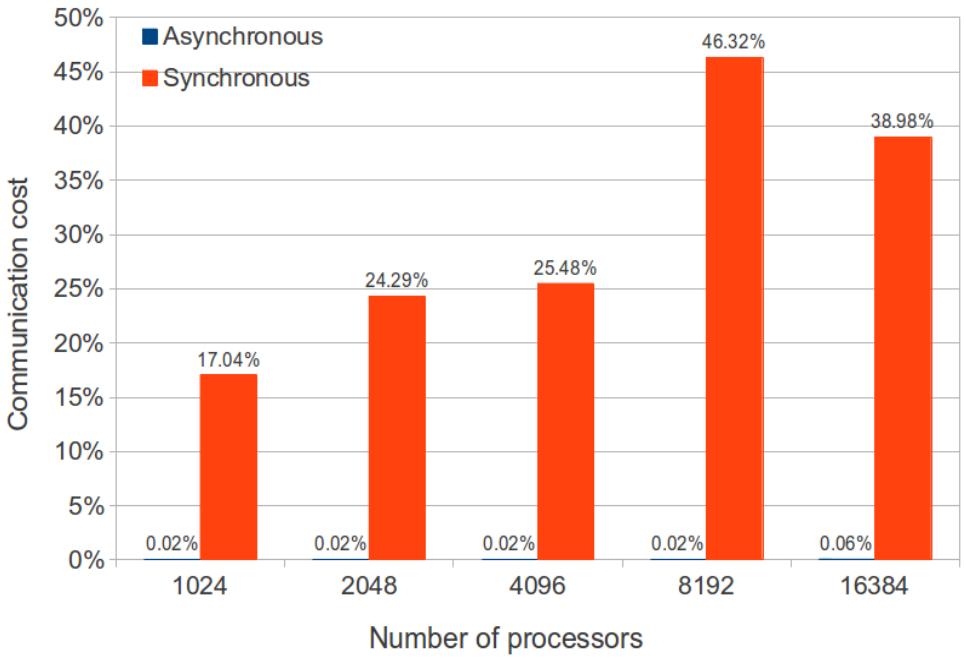
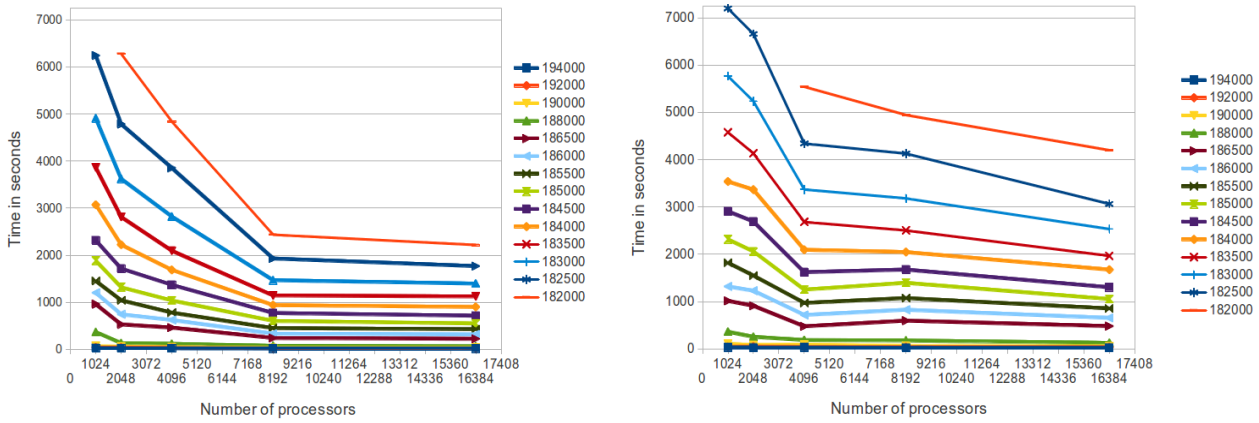


Figure 2.13: Communication cost on Stampede.



(a) Aynchronous migration

(b) Synchronous migration

Figure 2.14: Weak scaling results on Stampede.

nization cost is still high, which affects the communication cost. The communication cost using 16,384 cores on Stampede dropped to 38.98%, compared to that on Ranger (57%). The cost also increased as more processors were used, except for the case of 8192, possibly due to the layout of underlying topology of the InfiniBand network. Second, the computing speed is much faster on Stampede's Sandy Bridge processors. Consequently, the synchronization bottleneck stands out more obviously to negatively affect the speedup.

The asynchronous migration strategy also outperformed the synchronous version in the weak scaling test on Stampede, as shown in Figure 2.14. It also achieved better solutions in a shorter amount of time (Figure 2.14(a)), compared to that on Ranger (Figure 2.5(a)). More tight solution quality thresholds were achieved on Stampede, even when using smaller number of processors. Solutions better than 18,250 were found, but not on Ranger. Using 16,384 cores on Stampede, the solution quality improved by 1.26% (Figure 2.15) over synchronous runs, better than that on Ranger 1.09% (Figure 2.9), albeit higher variations on other cases due to more stochastic dynamics on Stampede. For the synchronous runs, using more than 8,192 processors did not reduce the execution time on Ranger (Figure 2.5(b)), while on Stampede, using more processors consistently reduced the execution time, as shown in Figure 2.14(b). In summary, the PGAP library performed consistently on Stampede and produced better results due to the faster CPUs and network in the system.

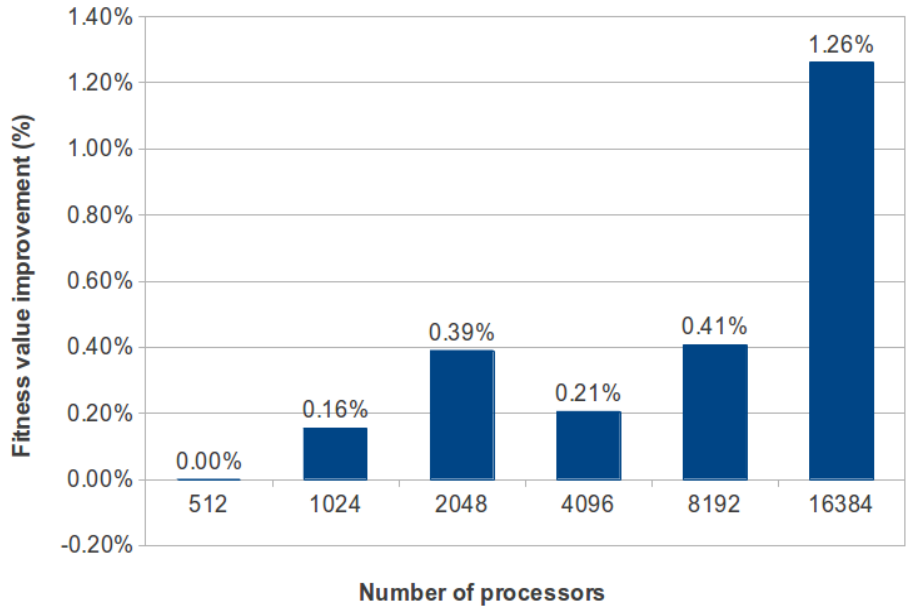


Figure 2.15: Solution quality comparison on Stampede.

2.4.8 Results on Heterogeneous Architecture

Coprocessor is one way to accelerate computation by putting hundreds of light-weight numerical processing units on a card and mounting it on a computing node with the node's host processors. The computation can be carried on such host + coprocessor architecture by either offloading computing intensive part to coprocessors or treating the host and coprocessor equally—so called symmetric computing. In PGAP, all the GA processes are identical and should use the symmetric computing model to run. However, the performance difference between the host and the coprocessor creates, sometimes a high degree of, variations that the PGA software has to handle. Apparently, PGA implementations using global synchronization is inefficient on such hybrid architecture: GA processes running on host processors have much faster pace and thus need to always wait for slower coprocessors to reach the communication barrier. PGAP, on the other hand, allows fast and slow GA processes to coexist and evolve and, thus, can fully leverage the computing power on coprocessor. Consequently, it is worth studying the algorithm behavior and computational performance of PGA in this configuration with the following evaluation scenarios.

1. Will solutions migrated from fast processes overwhelm local populations on MIC processes?

2. Will asynchronous migration code break given the high heterogeneity of computing and networking between CPU processes and MIC processes?

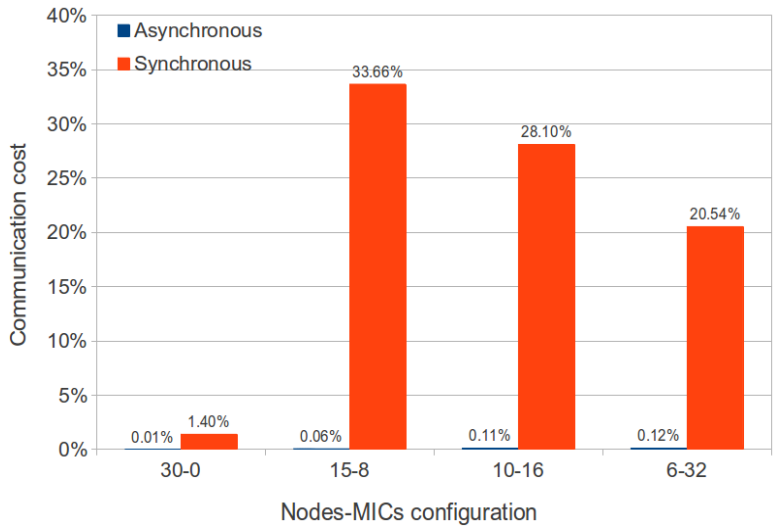


Figure 2.16: Communication cost with hybrid processor configuration.

3. How is PGA performance affected by inter-node and intra-node communications between CPU and MIC processors?

4. How does the use of coprocessor affect the configuration of PGA parameters?

On Stampede, each KNL node has 16 host processor cores and one Intel Xeon Phi MIC coprocessor with 61 usable cores. To handle different CPU speeds between the host and the MIC processor, an export operation in PGA is skipped when previous export operations are still pending.

In this evaluation, 240 cores are used to run the PGAP solver for any run, including 8 host processor cores per node. The number of nodes and coprocessor cores vary in the four configurations, each denoted $< num\_nodes >-< num\_MIC\_cores\_per\_node >$: 30–0 (30 nodes, no coprocessors); 15–8 (15 nodes, 8 coprocessor cores/node); 10–16 (10 nodes, 16 coprocessor cores/node); and 6–32 (6 nodes; 32 coprocessor cores/node). For each configuration, the communication cost, the number of iterations per second (iteration rate), and weak scaling are measured to test the computational performance. The number of local solutions are counted at each processor core to measure the numerical performance of PGAP in terms of the capability of generating local solutions given high heterogeneity among neighbors. A local new solution is counted only if it is better than the current best in the local population.

Figure 2.16 shows the communication cost results. In the 30–0 configuration without coprocessor involvement, the communication cost is the lowest since there is no CPU–MIC communication overhead. For the synchronous runs, synchronizing more coprocessor cards

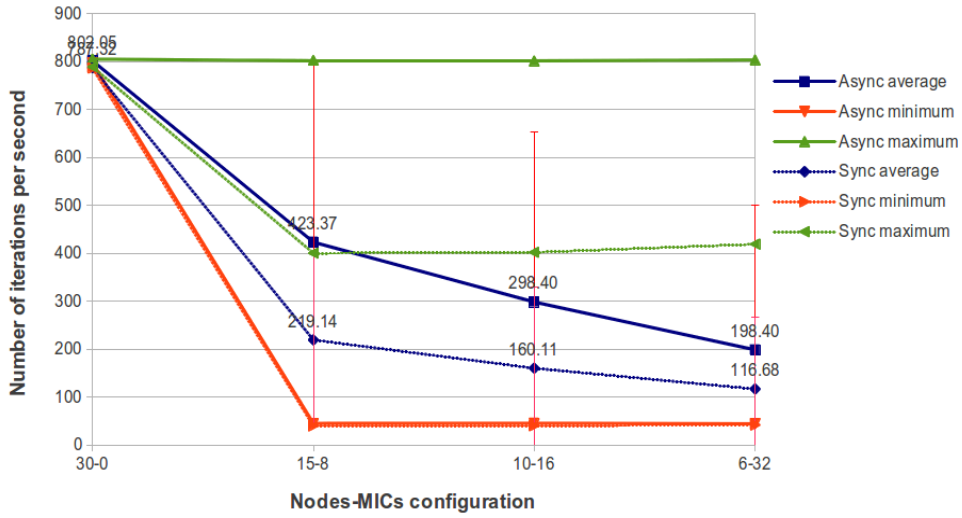


Figure 2.17: Iteration rate with hybrid processor configuration.

costs more than synchronizing more coprocessor cores in fewer number of coprocessor cards since coprocessor cores use one PCIe bus to communicate via the host CPU. A promising result from the communication cost test is the marginal cost using the asynchronous migration strategy. In asynchronous runs, the communication cost increased from 0.01% in 30–0 to 0.12% in 6–32 configuration, showing desirable communication performance, albeit the increase of slower coprocessor use.

Figure 2.17 shows the iteration rate as a major measure of the computational performance. The iteration rate on the host (maximum), the coprocessor (minimum), and on average are measured for both the synchronous and asynchronous runs. It is not surprising that the 30–0 configuration has the best iteration rate. The gap between the host and the coprocessor is apparent due to the CPU difference. The gap between the asynchronous and synchronous runs clearly shows the slowdown in the synchronous version, especially on the host iteration rate. Such slowdown is caused by the synchronization across both host and coprocessor.

Asynchronous migration was more dramatic in the weak scaling test, as shown in Figures 2.18 and 2.19. In the synchronous runs, the execution time needed to reach a solution quality threshold increased as more slow coprocessors were used. While in asynchronous runs, the 15–8 configuration exhibited particularly poor performance. The 10–16 configuration performed better than projected, though completed less numerical computing than the 15–8 configuration. This is more evident in Figure 2.19, which shows the solution improvement over synchronous runs. In the 15–8 configuration, the asynchronous run was worse than the corresponding synchronous run.

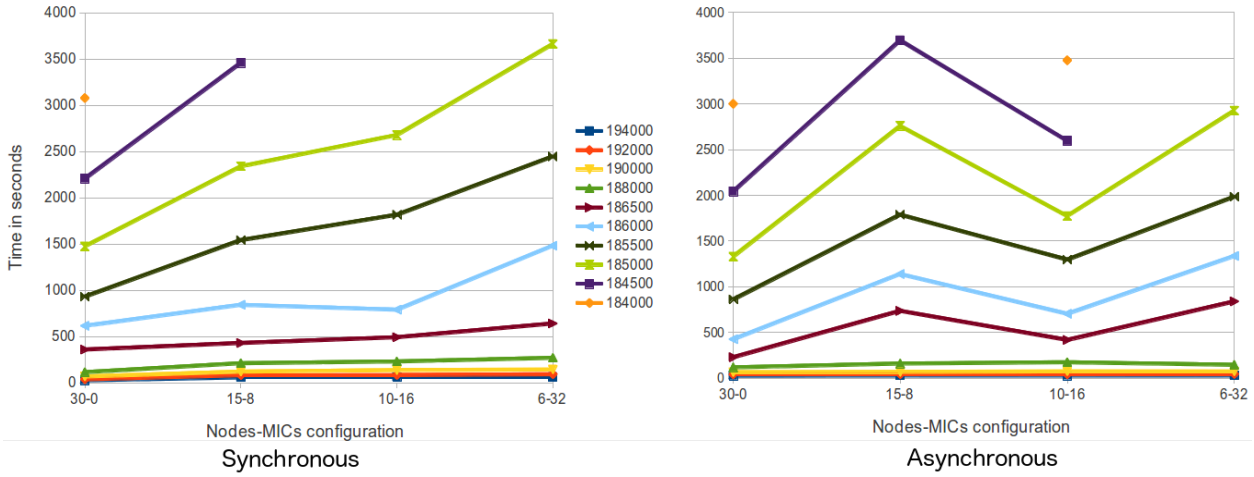


Figure 2.18: Weak scaling results with hybrid processor configuration.

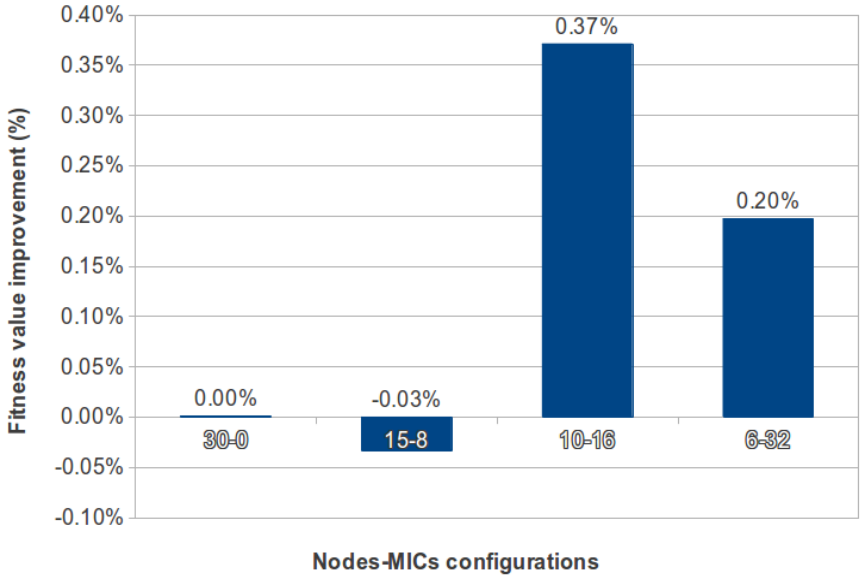
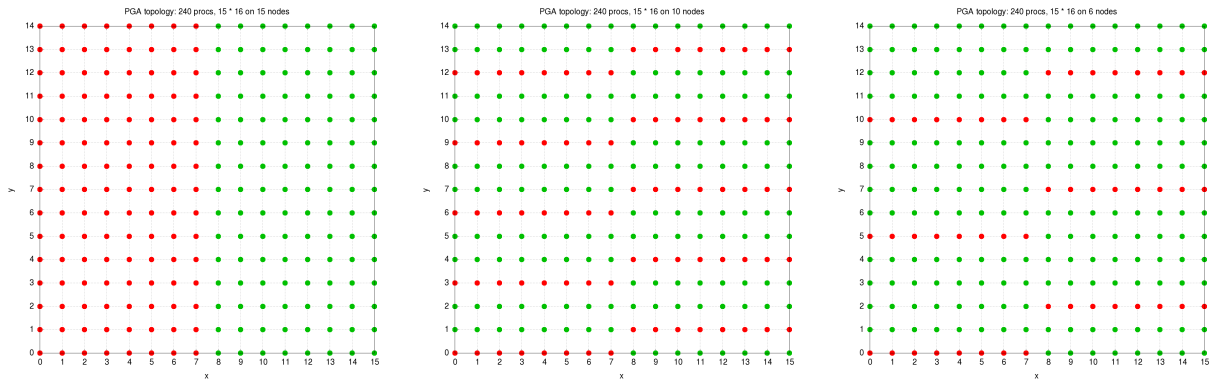


Figure 2.19: Solution quality comparison with hybrid processor configuration.



(a) 15-8 (b) 10-16 (c) 5-32

Figure 2.20: Topology of host and coprocessor configuration. Red points are host processor cores; green points are coprocessor cores.

After thorough investigation, the aforementioned variation is highly correlated to the neighborhood layout of the PGA topology. Figure 2.20 shows the topology for the three configurations with coprocessor involved. In terms of communication efficiency, host-host processor communication is the fastest, followed by host-coprocessor and coprocessor-coprocessor. In the 15–8 configuration, the left half are host processor cores (red points) and the right half are coprocessor cores. In the 10–16 configuration, every host processor core has at least one coprocessor core as neighbor, vice versa. In the 6–32 configuration, some coprocessor cores do not have host processor core neighbors. The 15–8 configuration performed poorly because the left host processor group proceeds much faster together, but the work done by the slower right half is then wasted. In the 10–16 configuration, slow coprocessor cores still contribute to the global evolutionary process. An unexpected benefit of this configuration is twofolds. First, coprocessor cores get more timely updates from their direct host processor neighbors than in the 15–8 configuration. Second, the migration delays from coprocessor neighbors allow host processor cores to spend more iterations to improve local population, a healthy delay effect mentioned by Hart et al. (1996). The 6–32 configuration unsurprisingly performed the worst because it had too many slow coprocessor cores involved and part of those cores without host processor neighbors were left behind.

The algorithmic performance was further evaluated by measuring the number of new solutions generated by each processor core. Figure 2.21 shows the average result in three phases of a PGA run, beginning (0–300 seconds), middle (300–1800 seconds), and end (1800–3600 seconds). The result is as expected. At the beginning, more new solutions are generated as the initial solution bound was quickly improved. Later, it becomes more difficult to find

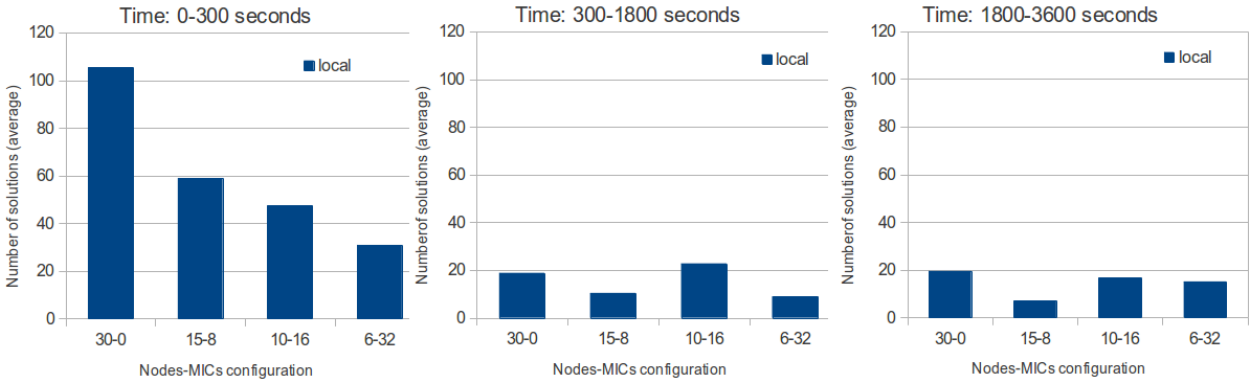
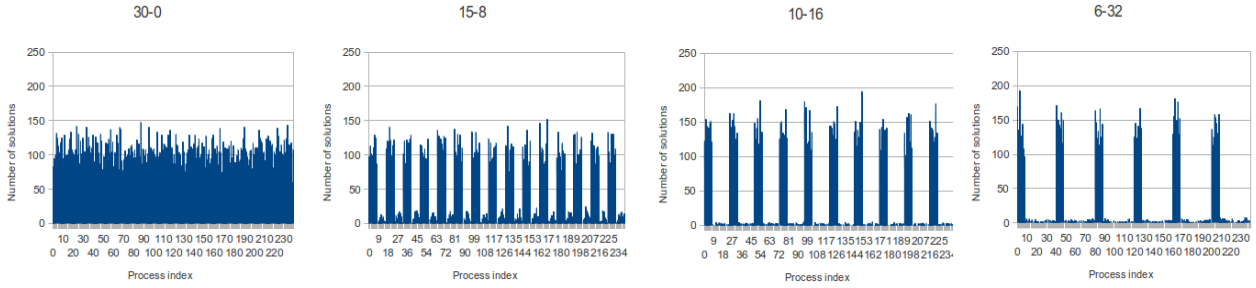
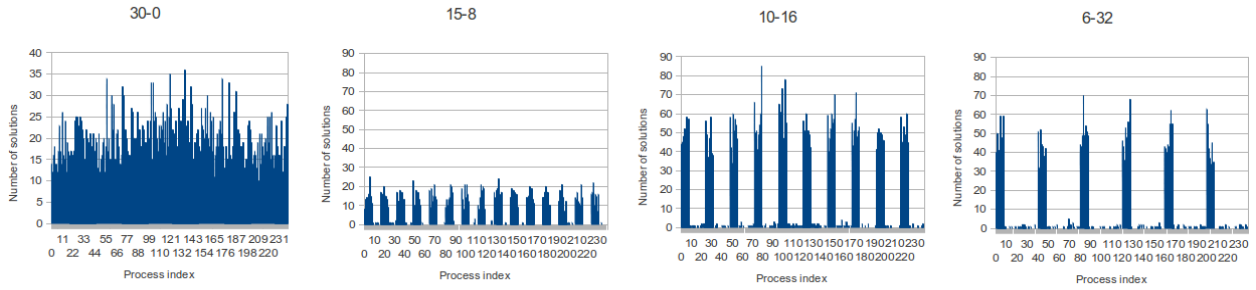


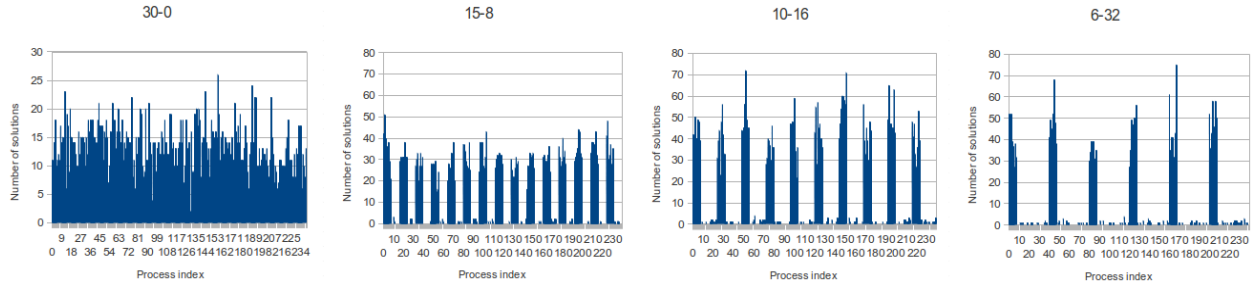
Figure 2.21: Average number of local solutions with hybrid processor configuration.



(a) 0-300 seconds



(b) 300-1800 seconds



(c) 1800-3600 seconds

Figure 2.22: Snapshot of the number of local solutions on each processor with hybrid processor configuration.

Table 2.4: Communication cost comparison.

	Asynchronous	Synchronous
Blue Waters	0.0137%	41.3893%
Stampede	0.0634%	38.9827%
Ranger	13.3966%	57.3898%

better solutions. The topology configuration of 15–8 caused slowdowns at later phases. Figure 2.22 provides a snapshot of each processor core, and clearly shows the numerical performance difference between the fast host processors and the slow coprocessors.

In summary, PGAP exhibits resilience on the coprocessor architecture. The communication cost is slightly higher than host-only configuration, but still marginal. Therefore, PGAP can fully leverage all the computing resources on such supercomputing architecture. This evaluation reveals that the topology, which defines the location of heterogeneous processors, can make significant difference in the numerical performance of PGA. This finding is interesting and will be investigated in future work.

2.4.9 Performance at Extreme Scale

As supercomputing steps into the extreme scale era (e.g, exascale (Lucas et al., 2014)), an application will be able to leverage millions of processors for a single run. To validate the computational scalability of PGAP at such extreme scale, similar performance tests were conducted on Blue Waters. Due to the limitation on the allocated CPU time, only weak scaling test using 16,384 to 131,072 integer cores and the asynchronous migration strategy were conducted. Figure 2.23 shows the communication cost on test runs using up to 131,072 integer cores on Blue Waters. Even at such large scale, the average communication cost is remains as low as 0.01676% on 131,072 cores, a convincing communication performance. The desirable communication cost partly attributes to Cray's low latency Gemini interconnect. Table 2.4 lists a comprehensive view of the communication cost on the three supercomputers used for scaling tests, based on the result of runs using 16,384 cores on each of them. Figure 2.24 shows the weak scaling result. This result shows consistent performance improvement by the reduced time taken to reach the specified solution quality thresholds. Solutions better than the tight threshold 182,000 were found in these large runs.

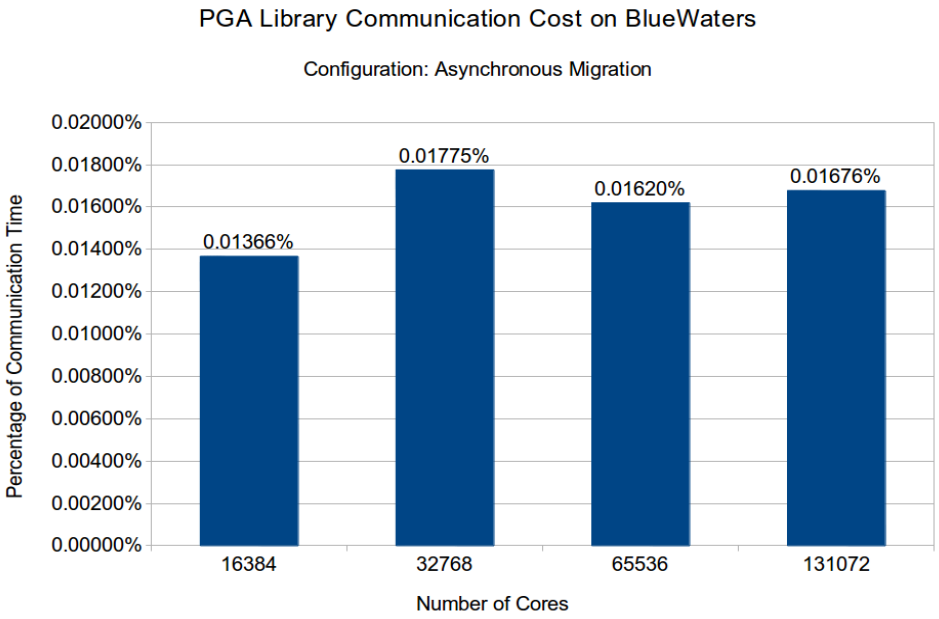


Figure 2.23: Communication cost on Blue Waters.

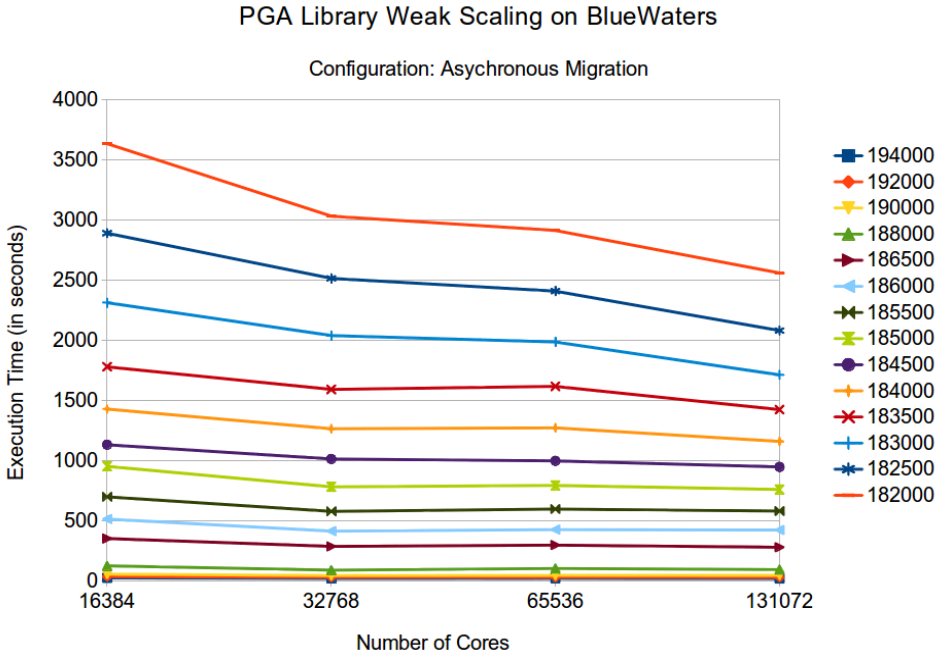


Figure 2.24: Weak scaling results on Blue Waters.

2.5 Summary

This chapter describes a scalable parallel genetic algorithm for solving large GAP instances by leveraging massively parallel computing. Since synchronizing massive cores imposes a significant performance penalty, an asynchronous migration strategy is developed to improve the numerical performance of PGAP. This strategy has three migration operators, export, import, and inject. By using buffer-based communication and non-blocking message passing between sending and receiving demes, migration communication can be overlapped with GA computation without adding any global barrier to synchronize demes.

Compared to the corresponding synchronous implementation, the experiments showed that PGAP significantly improves the communication and computation ratio, speedup, and the capability to explore the solution space efficiently. Superlinear speedups were observed in strong-scaling tests against large problem instances. The study of PGA's parallel efficiency is also important for understanding how to efficiently use computing resources to achieve a specified solution threshold given a particular problem. Cantu-Paz and Goldberg (2000) studied the speedup and efficiency of a simplified global parallelization-based PGA from a theoretical perspective and found interesting results on how to determine the optimal number of cores for solving a given problem instance. However, the study of parallel efficiency in coarse-grained PGA is more complicated and remains an open research problem because an optimal configuration of PGA may depend on the problem, the difficulty of reaching the specified solution quality, and runtime dynamics since PGA is a type of stochastic algorithm. By using the asynchronous migration strategy, the communication cost of PGAP is greatly reduced. As a result, each core is efficiently kept busy for local evolutionary computation. Computing resources can still be wasted if one deme largely repeats the work done by another deme. Similarity analysis on inter-deme populations, as part of our future work, would help the development of runtime strategies to avoid it.

Experimental results in the weak scaling experiment showed that PGAP exhibited desirable scalability for leveraging large population size enabled by using massive cores in solving large problem instances. To avoid runtime failure and the loss of good solutions observed in asynchronous PGAs, two buffer overflow problems are identified and addressed in PGAP by establishing two sufficient conditions for appropriate PGAP parameter configuration. The design of the asynchronous migration strategy is generic and independent of the targeted GAP problem. Therefore, the two sufficient conditions can be further applied as general guidelines for the development of asynchronous coarse-grained PGAs.

Three sets of experiments are designed to measure PGAP performance. The Ranger supercomputer was used to measure the PGA algorithm and its scalability, up to 16,384 processor cores. Results on Ranger showed that PGAP exhibited desirable scalability by achieving low communication cost. Near-linear and super-linear speedups on large GAP instances were obtained in the strong scaling tests. Desirable scalability to both population size and the number of processor cores was observed in the weak scaling tests. Using PGAP, large GAP problems were solved with desirable solution quality. This GAP solver is extended to a general PEA library.

The second set of experiment used a faster supercomputer, Stampede, to test the portability of the PEA library and its resilience on a hybrid computing architecture with coprocessor configuration. Better communication cost and scalability were observed on Stampede. The asynchronous migration strategy worked well and nicely fit into the hybrid computing environment coupling fast host processors and slower but large numbers of coprocessors. This experiment also revealed the importance of processor layout in the EA process topology, which can cause significant performance degradation if not properly configured.

The third set of experiments was conducted on Blue Waters, a supercomputer with 700k+ processor cores. The PGAP library scaled successfully to 131,072 integer cores on Blue Waters with marginal communication cost. As the first PEA library that scales to more than 100k processors, this published work (Liu and Wang, 2015) significantly improved the computational efficiency of evolutionary computation and provides a high-performance and scalable library for large optimization applications.

The findings presented in this paper have been incorporated in the PGAP software library. This library has been deployed on and tuned for the petascale Blue Waters supercomputer at the National Center for Supercomputing Applications (NCSA).

CHAPTER 3

TRANSFORMING POLITICAL REDISTRICTING THROUGH SCALABLE EVOLUTIONARY COMPUTATION

Ideas for large spatial optimization applications have become increasingly common as data availability has grown. Though desire and motivation is high, execution of these ideas lags the creativity that inspires them. There are two primary reasons for the mismatch. First, the execution of these large applications require more computing power (see Chapter 2), which often exceeds the level that is commonly available via desktop computers. Second, the ability to traverse these large solution spaces requires new search strategies that explicitly consider the spatial characteristics of the problem (see Chapter 3). It is only after we overcome these significant limitations that we will be able to gain insights into these large scale spatial optimization problems.

In this chapter, we provide an example of a large scale spatial optimization problem, namely, political redistricting. This application is particularly apt for our purposes because it not only exhibits the characteristic problems that are representative of the problems encountered in large scale spatial optimization, but does so at an extremely large and challenging scale. The number of potential solutions is astronomically large. That is, its solution space is sufficiently enormous that standard search techniques are entirely incapable of providing an effective and efficient means for identifying good solutions. Second, the application is inherently spatial and easily involves thousands to hundreds of thousands of intertwined spatial units. The potential solutions are embodied in maps that have strong spatial constraints, defined by the laws that govern redistricting in the United States. Providing solutions for the redistricting problem requires not only increased computing power but strategies for efficiently and effectively utilizing the increased computing power via both carefully crafted spatially cognizant heuristics. In addition, one can also achieve significant computational gains by designing algorithms that efficiently and effectively exploit massively parallel computing architectures.

We begin this chapter by introducing the application. In section 3.2, we situate our contribution with the previous literature, which has broached many different aspects of the problem including heuristic approaches, the consideration of spatial characteristics in

redistricting, as well as parallel evolutionary computation. In section 3.3, we formally define the redistricting optimization problem and describe how our problem formulation can be adapted to the mandates of the Supreme Court. Section 3.4 presents the evolutionary algorithm and its associated EA operators. Section 3.5 presents the parallelization of the proposed evolutionary algorithm and discusses the strategy for improving the efficiency of our algorithm computation in a massively parallel computing environment. Section 3.6 presents experiments and results to evaluate our sequential algorithm by comparing it with previously proposed heuristic approaches in the literature. The results of scalability analysis on the parallel EA are then presented. We also describe the application of our approach for studying the partisan gerrymandering phenomenon. Finally, in section 3.7, we discuss the implications of our approach for the practice of redistricting.

3.1 Redistricting

Political gerrymandering in U.S. dates back to at least 1812 when the term was coined by the *Boston Weekly Messenger* in an editorial cartoon depicting an eccentric election district that bore an uncanny resemblance to a salamander, complete with a head, arms, and a tail. That year, the Massachusetts state senate districts were redrawn under Governor Elbridge Gerry to favor the Democratic-Republican party by consolidating the Federalist vote. By all accounts, the redistricting was successful; while the Federalists won both the Massachusetts house and the governorship, the senate remained firmly in the control of the Democratic-Republican party. The practice of bolstering political power by carefully orchestrating voters into meticulously crafted districts is as old as the founding of the country. Even at its nascence, the irony was glaring; in what is touted as the greatest democracy in the world, politicians are able to essentially hand-pick their voters. It is not always, as we might presume in a democracy, the voters choosing their representative.

Gerrymanders are not looked upon favorably, and the practice of gerrymandering is obviously controversial, but regulating the practice has proven quite challenging. Despite general disdain for gerrymandering, its existence as part of American democracy is long-standing. While the Supreme Court has established guidelines and mandates to protect our democratic system of elections, there are so many ways to draw gerrymanders within the legal constraints that the phenomenon continues, scarcely impeded by regulation. The tools simply do not exist to conduct the types of analyses that are required by the Supreme Court to overturn gerrymanders. Plainly, adherence to democratic ideals is simpler to articulate than it is to ensure.

An under-explored vantage point stems from the realization that the Supreme Court's vision for regulation is closely aligned with a heavily computational analytic approach. The natural fit of a computational approach and the legal mandates portends that a significant gain to the science of redistricting will be realized via computational approaches. While others have sought to combine the insights of operations research approaches with redistricting, the proposed methods have been either computationally intractable or exhibited limited computational capabilities, unable to facilitate advanced quantitative study of the substantive problem at hand. We propose an effective evolutionary algorithm (EA) that embeds spatial characteristics such as contiguity, compactness, and a hole-free property into the algorithm design. Conventional EA operators, such as crossover and mutation, can disruptively break spatial constraints such as contiguity and hole-free requirements in redistricting plans, making multi-objective optimization even more difficult. We address this issue by efficiently incorporating the spatial configurations of the underlying problem within each EA operator, while still effectively utilizing EA's stochastic manner for exploring the solution space. As a result, our sequential EA outperforms other selected heuristic algorithms (e.g., simulated annealing, tabu, GRASP) in the literature for identifying better solutions in less time in our case study of redistricting in North Carolina and Maryland.

Our **P**arallel **E**volutionary **A**lgorithm for **R**edistricting, PEAR, is designed to tackle computational complexity and meet the challenge of generating a large number of districting plans for analysis. The scalability of PEAR to both the problem size and the number of processors is achieved by coupling the EA with a highly scalable parallel EA message passing framework. PEAR scales up to 131K processors and efficiently leverages the massive computing power made available from national cyberinfrastructures such as the Blue Waters supercomputer.[1] This parallel computing solution collectively evolves independent EA instances through periodic migrations. A non-blocking asynchronous migration mechanism eliminates the global communication barrier and the resulting significant communication cost in massively parallel computing environments. The scalability of our algorithm brings two desirable benefits for redistricting analysis. First, as the number of processors increases, our algorithm identifies solutions with higher quality. Second, larger numbers of processors provide a larger pool of good and feasible solutions, which in turn enables insightful statistical analysis of redistricting phenomena. To the best of our knowledge, PEAR is the first redistricting algorithm that is scalable to over a hundred thousand processors.

[1] http://bluewaters.ncsa.illinois.edu

3.2 Literature Review

Full Enumeration Approaches. The use of computers for redistricting was advocated many years ago (Vickrey, 1961; Nagel, 1965; Weaver and Hess, 1963). The 1960s saw the advent of early computer programs for this purpose (Nagel, 1965; Harris, 1964; Thoreson and Liittschwager, 1967; Gearhart and Liittschwager, 1969). The first explorations of this type focused on full enumeration approaches where the objective was to examine all possible redistricting maps in order to contextualize the current plan as simply one plan among a large number of other possibilities (Garfinkel and Nemhauser, 1970; Gudgin and Taylor, 1979; Papayanopoulos, 1973; Shepherd and Jenkins, 1970; Rossiter and Johnston, 1981). Because the redistricting problem is so computationally complex, the applications for this approach were for only *very* small redistricting problems. Indeed, the implemented methods are infeasible and not generalizable for any problem of practical size. In the 1960s, though enthusiasm was high, progress was essentially halted by computing technology that was insufficiently advanced to permit nuanced and helpful guidance for actual redistricting problems.

Simulation Approaches. The goals and research design for the latest round of redistricting research has followed the 1960s agenda closely. We are still enthralled by the full enumeration approach but are mindful that the computing capacity to achieve this goal still eludes us. As a result, research has re-focused on an obtainable and more feasible objective in the current computing environment. Instead of a full enumeration, the literature has shifted toward methods that are able to provide a large number of possible maps. The goal is to create a decision support system that allows one to contextualize a particular redistricting map by creating an ability to understand a range of possible redistricting maps for a particular geographic area.

Cirincione, Darling and O'Rourke (2000) attempt to derive a sampling distribution of maps. They utilize four different algorithms for generating plans: a contiguity algorithm, a compactness algorithm, a county integrity algorithm, and a county integrity and compactness algorithm. Their four algorithms all begin by randomly selecting a block group to serve as the "base" of the first district. In the next step, depending on the algorithm, some criteria are used to select an unassigned contiguous unit to expand the district. This process is repeated until the population of the district reaches a desired size. The next district begins with the random selection of a block group from those adjacent to one of the completed districts. In all, they examine 10,000 of these generated plans (2,500 from each of the four algorithms). Others have implemented a similar approach where districts are created by choosing an adjacent unassigned unit, considering the criteria of equal apportionment, contiguity, and

compactness, though they produce a considerably smaller number of simulated maps. Chen and Rodden (2013) appraise only a small and limited number of different plans (25, 250, and 200 different plans) to investigate questions of partisan bias in different states.

Integer Programming and Heuristics. Another strand of the literature capitalizes on the operations research literature on search heuristics. The gain here is rather than simply simulating maps that satisfy a minimal legal standard, one can search the space of legally viable maps and identify those that exhibit desirable characteristics. After all, the public, politicians, and the courts are interested maps that have a positive impact on democratic rule, not simply maps that unintelligently satisfy a minimum set of legal guidelines. In this vein, various integer programming approaches have been implemented. Mehrotra, Johnson and Nemhauser (1998) developed a branch-and-price methodology. Macmillan and Pierce (1994) implemented a simulated annealing algorithm for redistricting. Implementing a good search heuristic is non-trivial given the astronomically large number of feasible maps. Indeed, because of the computational complexity, Mehrotra, Johnson and Nemhauser (1998) were only able to apply their algorithm successfully to county-level data from South Carolina. Likewise, while Macmillan and Pierce (1994) restricted their algorithm to county data from Louisiana, Maine (16 counties, 2 districts), and New Hampshire (10 counties, 2 districts), they found the run times to be "intolerably long." Altman and McDonald (2011) implement four metaheuristics: simulated annealing, greedy search, tabu search, and greedy randomized adaptive search (Fleischer, 1995; Goldberg, 1989; Glover and Laguna, 1997; Feo and Resende, 1995). The authors state that their program "may yield a useful improvement over the starting map and may further enable the public to generate constitutionally viable plans." They also state that additional computational power may be needed and may be achieved in a parallel computing environment by using the *snow* package to run their R code. They do not present results from an actual redistricting application.

Spatial Characteristics. Preserving contiguity, compactness, political subdivisions and majority-minority groups requires proper representation of geographic information systems (GIS) data and maintenance of the proper spatial and geometric properties. In designing a search heuristic, care must be taken to store the necessary spatial and geometric attributes in appropriate data structures (e.g., graphs) since maintaining these spatial characteristics has an impact on virtually all of the operations. For instance, the contiguity constraint influences the choice of new solution generation and evaluation. Izakian and Pedrycz (2012) use a spatial scanner to search a map for clusters in their particle swarm optimization algorithm and had to check the contiguity of each cluster when using centroid to include regions. In

EA, traditional binary string genetic algorithm (GA) operators (e.g., crossover and mutation) tend to produce non-contiguous solutions (Liu, Cho and Wang, 2015). Unless using a penalty function, non-contiguous solutions have to be avoided by using either contiguity-preserving operators or repair functions (Mezura-Montes and Coello, 2011). Xiao (2008) summarized both approaches and applied them in a small redistricting case study. In Xiao (2008) and Xiao, Bennett and Armstrong (2002), the mutation operator is designed to maintain contiguity by changing either the shape or the location of a set of contiguous and moveable units. For crossover, since redistricting is similar to graph partitioning, crossover operators designed for graph partitioning, such as those in Galinier and Hao (1999), can be used in EAs for redistricting (Xiao, 2008). The crossover and mutation repair strategies in Xiao (2008), however, are inefficient for large redistricting problems because they involve costly and non-deterministic randomized trials and extra spatial and geometrical operations. Moreover, repairing contiguity is also likely to adversely affect gains in other objectives and constraints such as population deviation and competitiveness.

Parallel Evolutionary Computation. The structure of EA/GA is fortuitously highly adaptable for exploiting high performance and parallel computing resources for the stochastic and iterative evolutionary computation. Basic EAs/GAs evolve an initial population through a "survival of the fittest" rule via a set of standard stochastic operators, i.e., selection, crossover, mutation, and replacement (Holland, 1992; Fogel, 1997; Goldberg, 1989). In a parallel computing environment, a population may be naturally divided into a set of sub-populations (also called demes or islands) that evolve and converge with a significant level of independence. In this vein, various parallel EAs (PEAs) have been developed and applied to a broad and rich set of application domains (Alba and Tomassini, 2002; Konfrst, 2004; Huang and Rajasekaran, 2004; Hidalgo et al., 2010; Patvardhan, Bansal and Srivastav, In press; Muraro and Dilao, 2013). Alba and Troya (1999*b*) showed that PEA not only improves computational efficiency over sequential EAs, but also facilitates more extensive exploration of the solution space, resulting in a larger and better set of solutions. Ocenasek and Pelikan (2004) analyzed the complexity and scalability of parallel Estimation of Distribution Algorithms, a new paradigm of evolutionary algorithm. Though much efficiency can be gained with additional processors, one must also be wary of the substantially increasing communication costs that arise with additional processor cores (Jiang et al., 2004; Faraj, Patarasuk and Yuan, 2007). Synchronized migration has been linked to serious performance degradation on PEAs. As a result, asynchronous inter-deme interactions have been recog-

nized as desirable for designing scalable PEAs (Alba and Troya, 1999*a*, 2002; Hart et al., 1996; Lin, Punch and Goodman, 1994).

In summary, computational complexity is formidable and a major bottleneck for redistricting analysis. A full enumeration of plans is still not possible, and investigations at fine levels of geographic granularity have proven exceedingly difficult. How to traverse the space of possible redistricting maps is not straightforward. The availability of massive computing power provided by supercomputers provides an avenue for the development of scalable heuristics that are able to efficiently exploit massively parallel high-end computing resources to find better solutions and create a large number of unique feasible solutions for further statistical study. This is the task that we tackle here. We develop a highly scalable parallel evolutionary computation approach that intelligently explores the space of possible maps and efficiently extracts high quality feasible maps that satisfy flexible redistricting criteria.

3.3 Redistricting Analysis: A New Computational Approach

Drawing electoral maps amounts to arranging a finite number of indivisible geographic units of a study area (e.g., a U.S. state) into a small number of larger areas. For simplicity, call the former *units*, the latter *districts* or *zones*, and the study area *region*. Since every unit must belong to exactly one district, a districting map is a partition of the set of all units into a pre-established number of non-empty districts. The redistricting problem is an application of the set-partitioning problem that is known to be *NP*-complete and, thus, computationally intractable (Garey and Johnson, 1979). The total number of possible maps when drawing K districts using N units is a *Stirling number of the second kind*, $S(N, K)$, defined, combinatorially, as the number of partitions of an N-element set into K blocks (Keane, 1975). Even with a modest number of units, the scale of the unconstrained map-making problem is awesome. If one wanted to divide $N = 55$ units into $K = 6$ districts, the number of possibilities is 8.7×10^{39}, a formidable number.[2] At the census block level, California has 710,145 census blocks for 53 districts. Pointedly, the number of possibilities in actual redistricting problems is of staggering proportions, creating a prohibitively large computational problem. The large problem size and the computational complexity involved when working with fine-grained redistricting data has plagued the progress of tool development for redistricting analysis.

[2]The numbers are taken from the 55 counties and 6 congressional districts in West Virginia, USA.

3.3.1 Problem Formulation

Solution Space. The solution space in a redistricting problem is not characteristic of a rugged solution space. While the space landscape is hilly in the sense that it has the usual peaks and valleys, these peaks and valleys are not a rapid succession of precipices, but instead, a series of vast plateaus, and hence, not rugged in the traditional sense. These expansive plateaus manifest themselves throughout the landscape because many possible redistricting plans are extremely similar—it is readily evident that moving a single census block from one district to another does not induce much change, and there are a slew of such minor modifications to any redistricting plan. This idiosyncratic surface type coupled with the prohibitive size of the solution space behooves a tailored search algorithm that is able to make large moves within the solution space and has some mechanism to guide the search toward distinct areas in the solution space.

Problem Variables. The goal of the redistricting problem is to identify a plan that optimizes one or more selected objectives (e.g., competitiveness, safe districts, incumbent protection) while simultaneously satisfying legal constraints (e.g. contiguity and equi-populous districts). There are many ways to specify an objective function. A linear programming formulation of the problem might proceed as follows. We have a set of N geographic units, $u_1, u_2, \ldots, u_N$, that we wish to partition into a set of K districts/zones, $d_1, d_2, \ldots, d_K$. We can create an $N \times N$ adjacency matrix, $\mathbf{C}$, to indicate the contiguity of the various units, where the entries are defined as

$$c_{ij} = \begin{cases} 1 & \text{if unit } i \text{ and unit } j \text{ are adjacent or } i = j \\ 0 & \text{otherwise.} \end{cases}$$

for $1 \leq i \leq N$ and $1 \leq j \leq N$. The convention that $c_{ij} = 1$ for $i = j$ is adopted to simplify the checking of connectedness of districts. The population of the N units is denoted by $p_1, p_2, \ldots, p_N$. So, if the districts are equipopulous, then the population in each district would be the average population, $\overline{P}$, given by

$$\overline{P} = \frac{1}{K} \sum_{i=1}^{N} p_i.$$

Let $\mathbf{X}$ be an $N \times K$ matrix with elements, x_{ik}, denoting our decision variables. To specify a map, these variables are chosen for $1 \leq i \leq N$ and $1 \leq k \leq K$ so that

$$x_{ik} = \begin{cases} 1 & \text{if unit } u_i \text{ is assigned to district } d_k \\ 0 & \text{otherwise.} \end{cases}$$

The population in district k is then

$$P_k = \sum_{i=1}^{N} x_{ik}\, p_i \qquad \text{for } k = 1, 2, \ldots, K.$$

Constraints. We have constraints of at least three types.

1. Each unit must be assigned to exactly one district,

$$\sum_{k=1}^{K} x_{ik} = 1 \qquad \text{for } i = 1, 2, \ldots, N.$$

2. The maximum population deviation across all K districts is no greater than a specified value M. That is, for any two districts, d_i and d_j,

$$\mid P_{d_i} - P_{d_j} \mid \leq M \qquad \text{for } i, j = 1, 2, \ldots, K.$$

   To define the population deviation across all districts, we may formulate a population criteria as

$$p = \frac{\max\limits_{k}(P_k) - \min\limits_{k}(P_k)}{\overline{P}}\ . \tag{3.1}$$

   This measures the level of population deviation between the set of K districts. When the districts have identical population, $p = 0$. As their population increasingly deviates from one another, the value of p increases. In this formulation, it is possible for p to exceed 1. This occurs when the difference in population between districts is sufficiently large. In these cases, we set p to its maximum value, 1, which already represents an extreme population difference.

3. The units in each district must form a connected set. That is, each unit is accessible from any other in the set via transitions encoded in the adjacency matrix $\mathbf{C}$.

Other constraints include the hole-free property and compactness. The hole-free property requires that no district is wholly contained in another district. Compactness is encouraged and valued. However, since it is neither uniformly enforced by the courts nor strictly defined, it has taken on various specifications. A popular conceptualization, via an area–perimeter criterion, which compares the perimeter of a shape to the area of the shape, was first proposed by Ritter in 1882 (Frolov, 1975). With the area–perimeter compactness, a circle is the most compact shape and would have an area-to-perimeter ratio or compactness value of 1. The value of a simple area-to-perimeter ratio would vary with the size of the shape, but we can create a scale invariance by dividing the area by the square of the perimeter. We may also normalize the measure to have values in the $(0, 1]$ range by including π in the numerator. These changes result in one of the most widely used compactness measures in the area-perimeter class of measures, C_{IPQ} (Osserman, 1978), which is defined as

$$C_{IPQ} = \frac{4\pi A}{P^2}\,. \tag{3.2}$$

In our minimization formulation, we use $(1 - C_{IPQ})$ as the compactness measure. There are many ways to constrain shape or define compactness. An area-perimeter approach is only one such method. A shape may also be compared to a reference shape. We might use geometric pixel properties. Alternatively, other criteria may be based on the dispersion of elements in the area (Li, Goodchild and Church, 2013).

Objectives. Subject to the constraints above, we seek to optimize a single or multiple objectives. In a multi-objective scenario, we implement a weighted sum to incorporate multiple objectives in our solution fitness evaluation. The particular specification of the objective function is flexible, simply depending on the particular substantive interest. If one were interested in optimizing competitiveness, for instance, one might proceed as follows. Let D_k be the Democratic registration in district k and R_k be the Republican registration in district k for $k = 1, 2, \ldots, K$.[3] Assume that a district is most competitive when $D_k = R_k$. When all districts are considered, an overall measure of competitiveness in a map could be calculated as

$$f = T_p\,(1 + \alpha T_e)\,\beta, \tag{3.3}$$

[3]User may decide how best to measure partisan bias (with registration data, turnout data, presidential vote data, etc.), a substantive debate on which we take no position.

where

$$
\begin{aligned}
T_p &= \frac{1}{K}\left(\sum_{k=1}^{K}\left|\frac{R_k}{D_k+R_k}-\frac{1}{2}\right|\right), \\
T_e &= \left|\frac{B_R}{K}-\frac{1}{2}\right|, \qquad \text{for } 0 \le T_p \le 0.5,\ 0 \le T_e \le 0.5, \text{ and } \alpha, \beta \ge 0.
\end{aligned}
$$

Here, T_p measures competitiveness as a deviation of the Republican two-party registration from 0.50 in each district and T_e is a weighting factor, which captures the differential in the number of seats won by the two parties. In the formulation for T_e, B_R is the number of districts where Republican registration is larger than the Democratic registration; α defines the weight of T_e in the competitiveness measure; and β is a normalizing constant so that the value of f spans the range $[0, 1]$. For example, since $T_e \in [0, 0.5]$ when $\alpha = 1$, if we set $\beta = \frac{4}{3}$, then $f \in [0, 1]$.

This formulation with both the T_p and T_e components provides two layers of differentiation. By making T_e a weighting factor for T_p, we can simultaneously encourage the average registration differential, T_p, to be small while ensuring that this small differential is spread equally between the parties. Under our formulation for competitiveness, when $f = 0$, Republican registration and Democratic registration is the same and the number of districts where Republicans dominate and the number of districts where the Democrats dominate are identical.

3.3.2 Formulation Flexibility

The competitiveness, population, and compactness formulations are simply one of many ways to define constraints and objectives. One nice feature of our particular measures, shown in Equations (3.1), (3.2), and (3.3), is that the values are normalized so that they span the same $[0, 1]$ interval with 0 being the desired or optimal value. When the criteria are normalized in the same range, it simplifies the weight specification in a multi-objective function that encompasses measures reflecting competing interests that are ideally and simultaneously satisfied.

The criteria that are optimized are user-specified and may reflect any aspect of a redistricting plan that the user finds helpful to constrain or explore. In one specification, we could consider competitiveness as the sole objective, and population deviation and compactness as constraints. We may also consider optimizing (as a minimization problem) competitiveness,

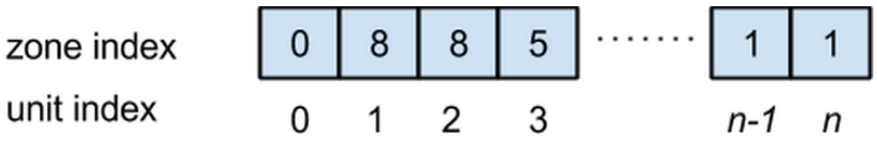


Figure 3.1: Chromosome encoding.

population deviation, and compactness while simultaneously treating population deviation as a constraint, by specifying a maximum threshold. While these choices are far from the only way in which one might guide such a computational model, they comprise reasonable examples of a real-world redistricting scenario that features the interplay of political interests and traditional districting principles. Once measures of the individual criteria are specified, the user may deploy any desired customized notion of how various criteria should be weighted. These specifications are modular, flexible, and customizable across a wide set of preferences, interests, and constraints.

3.4 Evolutionary Algorithm

Once the objective function is specified, we proceed to search for solutions that exceed a user-defined threshold of goodness. There are two main criteria for the design of a search algorithm. First, a stochastic element helps the search avoid being trapped in local optima. Second, since the solution landscape for redistricting maps is characterized by a series of vast plateaus, the algorithm must be able to make large jumps from one plateau in the solution space to another. Evolutionary searches are able to satisfy both of these criteria: 1) population-based heuristic search algorithms such as EA are always characterized by a strong random/stochastic element, which helps the search process avoid being trapped in areas with local optima; and 2) the crossover and mutation operator, in general, are intended to create large movements from one solution to the next. We develop an EA as a general redistricting solver. We employ efficient data structures and design effective EA operators in our EA to achieve these criteria for redistricting and handle spatial and non-spatial objectives and constraints. More importantly, the parallelization of our EA provides a scalable computational approach to employ a large number of processes that can simultaneously work on many plateaus and jump from one to another through inter-process communication.

3.4.1 Encoding and Data Structure

Similar to conventional binary string encoding, the basic data structure of the EA, the chromosome, is encoded as an integer array where each allele, indexed by the unit number, holds the zone (the term *zone* and *district* are interchangeably used hereafter) number that is assigned to each of the geographic units. An example is displayed in Figure 3.1, where we can see, for instance, that both geographic units 1 and 2 are assigned to zone 8 while geographic unit 3 is assigned to zone 5. Every geographic unit is assigned to exactly one zone.

In addition to the binary string encoding of a solution/chromosome, we maintain several spatial data structures to enable spatial constraint checking and operations. We store two graphs with auxiliary indexing data: the unit graph and the zone graph. Consider an example redistricting problem derived from partitioning the 87 county geographic units in Minnesota into its 8 congressional districts. The unit graph, which is also the county map, is shown on the left in in Figure 3.2.[4] We first convert the geographic data/map into a network graph with vertices and edges where the vertices are the counties and an edge exists whenever two geographic units are adjacent. We also define a virtual unit 0 which is a polygon derived by subtracting the shape of the region (state) from a rectangle that contains the region border. Unit 0 is particularly useful in identifying units on the region border. The adjacency structure is shown on the right in Figure 3.2 where the blue lines indicate rook adjacency and the red lines indicate queen adjacency (unit 0 is not shown). Since both rook and queen adjacency fall within the legal definition of contiguity, we encompass both in our adjacency matrix. A zone graph, Gz, which is a $K \times K$ 0–1 matrix, stores the adjacency of the districts. The zone graph is used to check if a district is contained in another district, thus creating a "hole" in the redistricting map. Similar to the definition of unit 0, zone 0 is a virtual district on the zone graph that has only one unit, which is unit 0. To handle large problems, the unit graph is stored as a linked list, instead of a 2-D array (the number of edges of a planar graph is a linear factor of the number of vertices). Auxiliary data structures are also used to help accelerate sorting operations on unit attributes loaded from GIS data.

Operations on the aforementioned data structures form the basis for handling the spatial configurations of the redistricting problem in the EA operators. These operations include, for example, finding neighborhood units, identifying the border units of a district, contiguity checking, hole checking at the district level, and compactness updating. We now describe

[4]Our algorithm is intended to be utilized at fine levels of granularity (e.g., units being census tracts or census blocks), however, because it is difficult to visualize these small units, we illustrate our EA at the county level in this section.

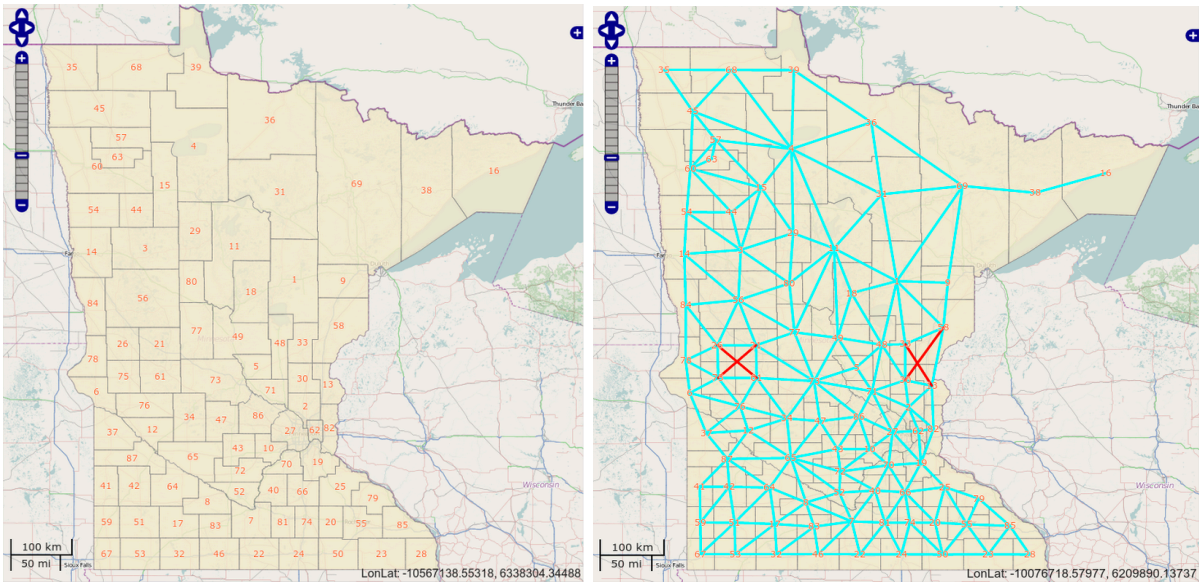


Figure 3.2: Neighborhood representation.

how these operations are designed efficiently to prevent excessively hampering the EA's numerical performance.

Contiguity checking. As we form districts, we must ensure that all formed districts are contiguous and that all the units within each district remain contiguous. At the district level, the small size of the zone graph makes this check simple and straightforward. At the unit level, the check is non-trivial since the associated computational cost increases dramatically as the number of units grows. King et al. (2012) described this cost and proposed a geograph approach to resolve the performance issue. Rather than performing the check after a new solution is generated, we adopt a strategy that integrates contiguity checking within the EA operators. The correctness of our strategy is guaranteed by the following two rules:

1. An initial solution satisfies the contiguity constraint; and
2. Any changes to the solution created by unit movement (from one district to another) does not lead to a non-contiguous solution.

While the second rule seemingly imposes a strong condition for the design of EA operators (e.g., crossover and mutation) to satisfy, we find that maintaining contiguity within our EA operators avoids the expensive solution-level contiguity checking and the often failed contiguity repair operation. The capability of traversing the solution space is still maintained by introducing randomization at the unit selection stage of each EA operator, as described in section 3.4.2. To determine whether a move breaks contiguity, we check if all of the units in a

district remain contiguous when a subset of units in the district is removed. The pseudocode for our contiguity checking is shown in Algorithm 3.1.

Algorithm 3.1: Contiguity checking.

```
1  U: set of units to be removed
2  S: a solution
3  Check that at least one unit in U is on zone boundary
4  V = {}  // units that are neighbors of U and in the same zone
5  foreach u ∈ U {
6    foreach neighbor, v, of unit u {
7      if(v is in the same zone in S)
8        V = V ∪ {v}
9    }
10 }
11 G = {}; // connectivity graph as adjacency list
12 foreach u ∈ V {
13   G[u] = {};
14   foreach neighbor v of u {
15     if(v ∈ V)
16       G[u] = G[u] ∪ {v}
17   }
18 }
19 Starting from any unit, count the number of connected nodes on G,
20 c return(c > 0 and c == length(V));
```

Hole checking. Before a new solution is generated for evaluation, we must ensure that holes (i.e. when a zone is contained within another zone) are not created in the redistricting map. The hole checking function examines the zone graph matrix for zones with only one neighbor. If the neighbor of such a zone is not zone 0, the hole-free requirement is violated. If the only neighbor is zone 0, this zone is isolated, violating the district contiguity requirement. This case is unlikely unless the EA operations are improperly coded.

Compactness calculation. In our formulation, the worst compactness value among all districts in a solution determines the compactness for the redistricting map as a whole. To calculate the compactness of an individual district efficiently, we do not store the geometric information (i.e., the shape and coordinates) for each individual unit. Instead, we store the area and perimeter of each unit, and border length between any two units. This allows us to calculate the area of each district by summing the areas of each unit in the district. The perimeter of a district is the summation of the border length between any two units in which only one unit belongs to the district. Calculating the perimeter for all districts can be efficiently accomplished with a dynamic programming approach where each unit–unit border is visited only once. This approach avoids expensive operations involving geometry such as

a spatial join. Instead, the only cost comes from the preprocessing of the shape of each unit, which is completed once before the execution of the EA.

3.4.2 Evolutionary Algorithm Operators

An EA algorithm includes a set of operators that generates the initial population, produces new solutions from parent solutions, and evaluates them. For redistricting, the objectives and constraints are tightly coupled with a districting map's spatial configuration. As a general principle, each move in our EA operators maintains contiguity while still using randomization to traverse the solution space. This strategy ensures greater numerical efficiency, an important consideration in implementing these operators.

Population initialization. The EA algorithm begins by generating a set of initial solutions to form the initial population. To generate an initial solution, we first randomly select K seed units and then expand them to K districts by iteratively including a random number of direct neighbors. This iterative unit-expanding method guarantees contiguity. Seeding proceeds either via administration boundary or by region border. In the first strategy, K seeds are selected such that each belongs to a different higher-level administration boundary (e.g., county for voter tabulation district level redistricting). This strategy generates more compact solutions but may lead to maps with holes (in which case, we simply ignore and begin the process again to avoid expensive repair procedures). The second strategy randomly selects K seeds on the region border, which guarantees the hole-free requirement since districts touching the region border cannot be completely surrounded by any other district. In the algorithm, a probability value is specified for each strategy to be applied in population initialization. It is worth noting that the seeding and expanding strategy is a common strategy used in construction-based heuristic algorithm design, e.g. in geographic analysis (Li, Church and Goodchild, 2014), spatial optimization (Brookes, 2001), and clustering (Hamerly and Elkan, 2002).

The second strategy for initial solution generation is illustrated in the series of maps shown in Figure 3.3. We begin by randomly choosing 8 geographic edge units to be our seeds. An example is shown in the leftmost map of Figure 3.3. We then expand these seeds by selecting contiguous units (center map in Figure 3.3) until we have all 8 connected districts composed of contiguous geographic units in each (rightmost map in Figure 3.3). The result is inserted into the initial population as a feasible solution. This process is repeated until the initial population is fully populated. The number of solutions in the initial population can be set to any value. The default is 200. The corresponding pseudo-code appears as Algorithm 3.2.

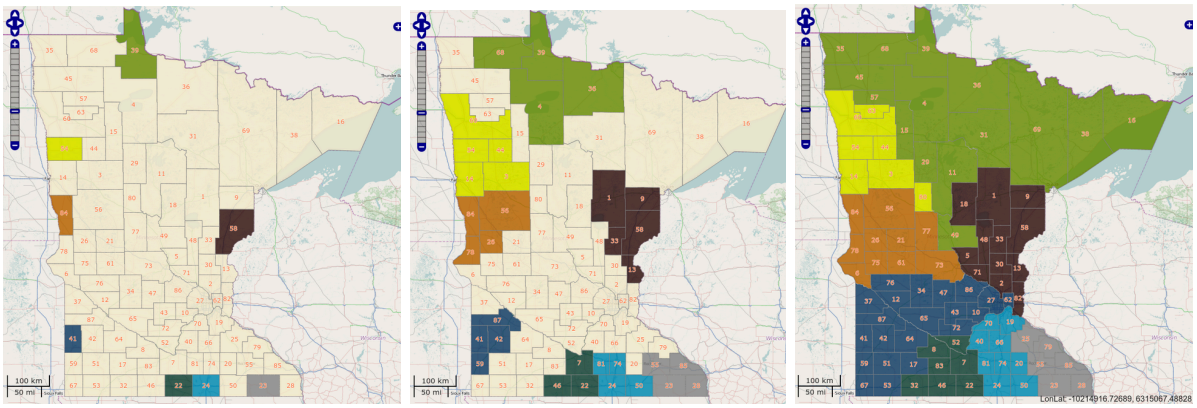


Figure 3.3: Solution initialization.

Algorithm 3.2: Generating a feasible contiguous solution.

```
1  // choose K initial seeds
2  Seeds = ∅
3  Get the list of units neighboring unit 0, N_v
4  while(|Seeds| < K) {
5    Randomly select a unit u from Nv
6    if(u is not selected before)
7      Seeds = Seeds ∪ {u}
8  }
9  // generate a solution from seeds
10 int soln[n] // initialized to zeros
11 Pool = Seeds
12 while(Pool ≠ ∅) {
13   Pop a unit u from Pool
14   Get the neighbors of u, N_u
15   foreach unit v in N_u {
16     if(soln[v] = 0 and v ∉ Pool)
17       Pool = Pool ∪ v
18       soln[v] = soln[u]
19   }
20 }
21 return soln
```

Fitness evaluation. The fitness value of a solution is the same as the objective function value, which is flexibility configured to be single- or multi-objective via a weighted sum formula. Contiguity and population equality must be satisfied by law.[5] We incorporate this legal requirement by considering a solution to be feasible if its population deviation is below a specified threshold. We operationalize this concept by calculating an unfitness

[5] *Reynolds v. Sims*, 377 US 533 (1964), created the mandate of one man, one vote. Though strict equality is not demanded, there is no *de minimus* deviation that is allowable (*Karcher v. Daggett*, 462 US 725 (1983)), and minimizing the population inequality is a legal requirement.

value for each of our solutions. The unfitness value is zero when the population equality across the set of districts is within the threshold value and is otherwise the value of the population deviation measure. Since contiguity is required for all solutions, non-contiguous solutions are discarded. In this way, our formulation incorporates constraints that, unlike the objective, must be satisfied by any solution. Each solution has a fitness and an unfitness value, which facilitates flexible replacement strategies (either through a penalty function or other algorithmic logic).

Mutation. The spatial mutation operator is designed to maintain contiguity while leveraging randomization to select units for district reassignment. It has two procedures:

1. *Shift* moves a number of units from one district to a neighboring district. To ensure that a shift does not violate contiguity, the selected units include at least one unit on the boundary of the sending and receiving district.

2. *Mutate* makes a sequence of shifts to balance metrics such as population deviation. This sequence may have one or more cyclic shifts.

Algorithm 3.3: Shifting mutation.

```
1   // Multi-unit mutation process that runs K times to make a chain of shifts
2   mutate(solution) {
3     Generate a random sequence s of size K using the Fisher-Yates algorithm;
4     foreach zone z ∈ s in solution {
5       if (z is a source zone of previous shifts) continue
6       if (population in z is below a threshold) continue
7       Randomly select a dstZone from neighboring zones of z
8       shift(z, dstZone)
9     }
10  }
11  // Select a subset of units from the source zone to the destination zone
12  // srcZone: source zone to send the selected units
13  // dstZone: destination zone to receive the selected units
14  shift(srcZone, dstZone) {
15    Randomly select up to two adjacent units in srcZone bordering dstZone;
16    subZone = selected units;
17
18    while |subZone| < maxMutUnits {
19      Find neighbor units U of subZone;
20      Generate a random number q ∈ 1, 2, ..., |U|;
21      Randomly choose a subset U' ⊂ U, where |U'| = q;
22      subZone = subZone ∪ U'.
23    }
24    dstZone = dstZone ∪ subZone;
25    srcZone = srcZone − subZone;
26  }
```

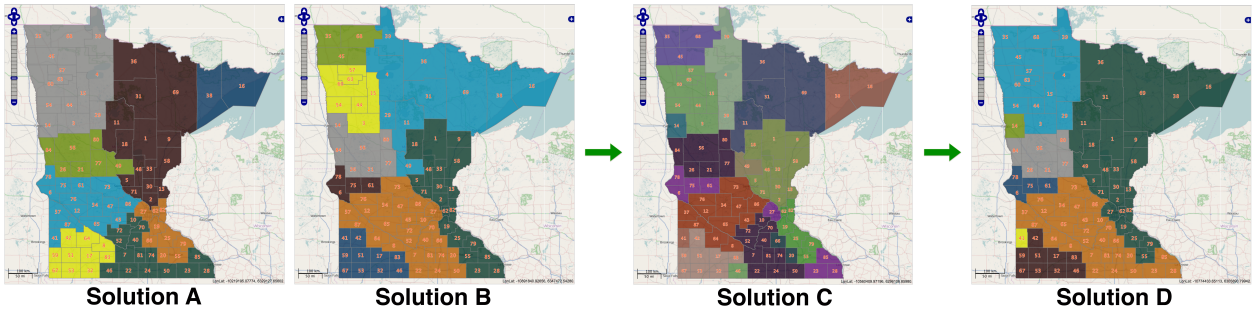


Figure 3.4: Crossover operator.

In *shift*, we randomly select a boundary unit and then expand it randomly to form a set of units to be moved. In *mutate*, we randomly choose a sequence of districts and randomly choose the number of cyclic shifts. The spatial mutation algorithm is outlined in Algorithm 3.3. The parameter $maxMutUnits$ is set at runtime. Larger redistricting problems should use a larger value in order to avoid small and ineffective moves.

Crossover. Just as the conventional mutation operator is not suitable for spatial configuration, the binary string crossover operators must also be modified to take spatial considerations into account. Our crossover operator, shown in Algorithm 3.4, overlaps two redistricting maps (e.g., solution A, the first map in Figure 3.4, and solution B, the second map in Figure 3.4), resulting in a set of intersected subzones/splits, shown in the third map, solution C, in Figure 3.4. Solution A has districts $A_1, A_2, \ldots, A_8$. Solution B has districts $B_1, B_2, \ldots, B_8$. The third map will have between K and K^2 split labels, C_{ij}, where $i \in \{A_1, A_2, \ldots, A_8\}$ and $j \in \{B_1, B_2, \ldots, B_8\}$. That is, each split label is formed from exactly one district from solution A and one district from solution B. However, depending on the district shapes, different splits may share a common label. We subsequently relabel them as new splits.

The splits shown in the third map form a new split-level redistricting problem. The number of districts in the new problem is still K. We form a new solution by treating splits as units via the same population initialization strategies which will combine these splits into a feasible solution of K districts. Since each unit belongs to only one split, it is straightforward to convert this split-level solution to the unit level. The new solution is then returned as the output of the crossover operator. The last map in Figure 3.4 shows Solution D, the map after the crossover. It is worth noting that, while the crossover is able to generate large movements to new solutions, likely in different parts of the solution space, our empirical tests indicate that this process may be sufficiently disruptive to the desirable

attributes of the underlying maps that the resulting child map may not be of higher quality than the parent maps. Solution D seems to represent such a case. Accordingly, we apply the crossover operator with relatively low probability. This permits larger moves in the solution space while maintaining reasonable numerical performance and EA convergence.

Algorithm 3.4: Crossover.

```
1  crossover(solution1, solution2) {
2    // Identify splits by assigning unique index to each split
3    foreach unit u {
4      Calculate split label as z1z2, where z1 is the zone index of u
5      in solution1, and z2 is the zone index of u in solution2
6    }
7    Group units with the same split label into the same split, form set Splits
8
9    // handle the case: two geographically separated splits are assigned with same index
10   newSplits = ∅
11   foreach split s ∈ Splits {
12     while s ≠ ∅ {
13       Find a unit u in s, construct a spanning tree in s, rooted at u
14       Form a new split s' to include all of the units on the spanning tree
15       newSplits = newSplits ∪ s'
16       Remove all the units in s' from s
17     }
18   }
19
20   // solve the split-level redistricting problem
21   Construct split graph (rook and queen neighborhood) from newSplits
22   Check holes and repair
23   Formulate a new redistricting problem
24   Generate a solution using the 2nd seeding strategy in the population initialization operator
25   Convert split-level solution to unit-level solution soln
26
27   return soln
28 }
```

After the mutation and crossover, a new solution, if its population deviation is above the defined threshold, goes through a fine-tuning procedure that examines the population difference between any pair of connected districts by checking the zone graph. If necessary, some units are exchanged between a pair of districts to balance the population in a randomized way. The last step in an EA iteration is to evaluate the resulting solution and apply a replacement strategy to update the population. Both the fitness and the unfitness are considered in the replacement strategy. An existing solution with the worst fitness or unfitness is replaced if a new solution is better. If a new solution is better than the current best, it becomes the elite solution. Therefore, our EA is a *steady-state* EA Goldberg (1989) in which each iteration selects two parents to generate one child for replacement.

3.5 Parallelization

While the sequential EA algorithm efficiently incorporates the spatial characteristics of the redistricting problem in its formulation, the solution space for the redistricting problem is idiosyncratic, astronomically large, and characterized by sprawling plateaus. Traversal of this type of solution space requires significant computation that can be aided by employing parallel computing on a large number of processors. PEAR incorporates a migration strategy that exchanges solutions between any two directly connected islands (processes) at regular intervals. To ensure each evolutionary process is independent, each island uses a unique random number sequence generated by SPRNG (Mascagni and Srinivasan, 2000). The use of SPRNG in the message passing (MPI) model ensures that no two processes would repeat the same random number sequence (starting two processes with different seeds is not enough since they may use the same random number sequence) and, thus, exhibit similar search paths when running independently even without external random noise. This is particularly important when a large number of processes run in parallel. Liu and Wang (2015) develop a scalable PGA library with a suite of non-blocking migration operators (i.e., *export*, *import*, and *inject*) to eliminate the need for a global barrier in migration communication. PEAR extends this library to enhance the parallelism control by handling application-level sending and receiving buffers for non-blocking message passing. The regular migration of elite or random solutions on all of the islands enables simultaneous exploration of a large number of solution space plateaus as well as movement from one plateau to another through elite solution propagation and the collective but independent evolutionary searches surrounding these elite solutions. The asynchronous migration strategy maximizes the overlapping of computation and migration communication and removes prohibitively costly global synchronization in a massively parallel computing environment. Table 3.1 lists the configuration of our EA and PEA.

In Liu and Wang (2015), the relationship between the configuration of the PGA parameters (i.e., migration intervals, migration rate, and topology attributes) and buffer sizes is established based on the underlying message passing communication library and supercomputer interconnect characteristics to avoid buffer overflow issues at the system and application levels. It also provides explicit programming control on the parallelism of the import operator by configuring the import buffer size and the interval for invoking the import operator in GA iterations. Further experiments, however, observed MPI communication layer failure when scaling from 16K processors on the Stampede supercomputer to larger numbers of faster processor cores on supercomputers such as Blue Waters (with more than 700K integer

Table 3.1: PEAR parameter settings.

Parameter	Value
Population size per deme	200
Initial population	80% by region border, 20% by administration boundary
Selection	Binary tournament
Spatial mutation	Probability: 0.95; $maxMutUnits$: 15
Spatial crossover	Probability: 0.05
Repair	District population balancing on infeasible solutions
Replacement	Replacing the worst (fitness) or the unfittest
Elitism	Yes
Stopping rules	No solution improvement, bounded solution quality reached, fixed number of iterations, execution walltime
Process topology	2-D Torus
Number of islands per processor	1
Process connectivity d	4 (number of direct neighbors of each process)
Migration rate r	2 (number of solutions sent to neighbors in each export)
Export interval M_{expt}	50 (number of iterations between two consecutive exports)
Import interval M_{impt}	25 (number of iterations between two consecutive imports)
Sending parallelism	2 (number of exports to invoke before waiting for their finish)
Sending buffer size $K_{sendbuf}$	2 solutions. Actual memory requirement is $(2 \times n \times 4 + \text{buffer\_overhead})$ bytes
Receiving buffer size K_{impt}	8 solutions. Actual memory requirement is $(8 \times n \times 4)$
Multi-objective fitness function	Scalability test: $0.8 \times$ competitiveness + $0.2 \times$ population_deviation + $0.3 \times$ compactness

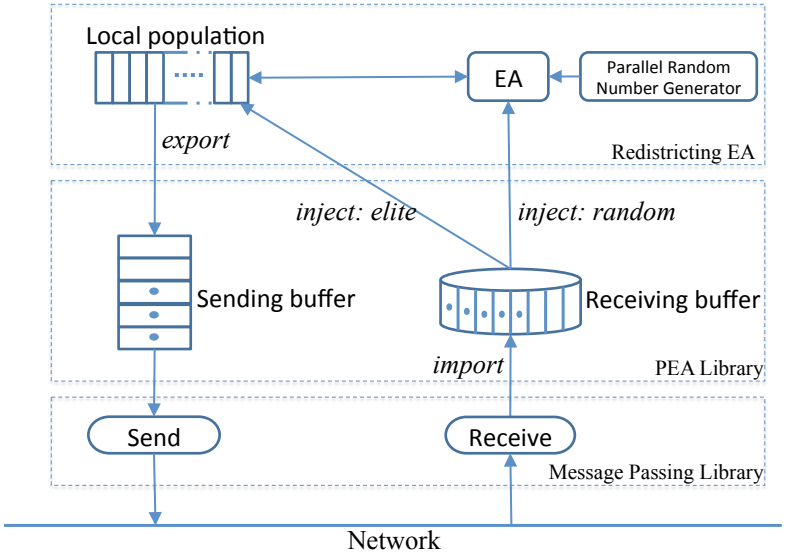


Figure 3.5: Enhanced PEA framework.

cores). With more processors, the outgoing message buffer, controlled by the MPI, experienced buffer overflow as message sending from the export operator become seriously skewed among PGA processes due to the outpaced runtime delays from numerical operations and non-blocking sending and receiving. Consequently, we extended the PGA library to manage the sending buffer at the application level and explicitly specified the degree of overlapping between EA iterations and message sending. Our enhanced PEA framework is shown in Figure 3.5. The improved library has been tested and scales well on the Blue Waters supercomputer without failure and with marginal communication cost. This extension has minimum impact on PEAR's numerical performance. The communication cost remained consistently at around 0.015% with 16,384 processor cores on Blue Waters. Such scalability provides a dramatic increase in numerical capability, especially notable and consequential since experiments using synchronous migration exhibited an increasing communication cost of 41.38%.

3.6 Evaluation

We evaluate our computational approach from three perspectives. First, we examine our base sequential algorithm by comparing its solution quality to that obtained by other heuristics.

This comparison allows us to evaluate the effectiveness and numerical efficiency of our spatial EA operators. Second, we evaluate the parallel component by examining the scalability of the PEAR algorithm by quantifying the numerical work that is completed by different numbers of processors. Third, we take note of the large number of feasible solutions that are generated in the scalability test, consider a statistical analysis in the redistricting context, and discuss how the maps inform an investigation of partisan gerrymandering.

3.6.1 Implementation and Case Study

Our EA is implemented in ANSI C. It can be compiled on Linux, OS X, and Windows as a standard *makefile* project. PEAR uses MPI non-blocking functions (i.e., *MPI_Ibsend()*, *MPI_Iproble()*), and regular *MPI_Recv()* for asynchronous migration. It uses the C SPRNG 2.0 library to provide a unique random number sequence for each MPI process, which is necessary for running a large number of EA iterations. The evaluation of our sequential EA is tested on the ROGER supercomputer. The scalability of our code is tested against MVAPICH2 with up to 16,384 cores on the Stampede supercomputer and the Cray MPI with up to 131,072 integer cores on the Blue Waters supercomputer. Each node on ROGER is configured with the Intel Xeon E5-2660 processor (2.6GHz, 20 cores/node). Stampede employs the Intel Sandy bridge processor (2.7 GHz, 16 cores/node). Blue Waters utilizes the AMD Bulldozer processor (2.3 GHz, 32 integer cores and 16 floating point cores per node).

Without loss of generality, the data for the empirical evaluation of our algorithm come from North Carolina and Maryland. Our evaluation is at the voter tabulation district level (VTD). GIS census data are available online for all U.S. states while election results are available at the VTD level for some states. These data include the shape, population, election, and party registration information for each district. Gien their legal and political history, both North Carolina and Maryland provide interesting redistricting applications. Our PEAR solver, however, is general and takes three types of inputs: the GIS data, the rook and queen neighborhood matrix, and information needed for compactness evaluation (i.e., area and perimeter information as well as the unit–unit border length information). The latter two types of input can be obtained using open source GIS libraries such as PySAL (http://pysal.org) and GDAL (http://gdal.org).

3.6.2 Comparison with other redistricting tools

To evaluate the performance of our EA algorithm and the effectiveness of the EA operators, we compare our sequential PEAR algorithm to the BARD redistricting software (Altman and

Table 3.2: Performance of sequential heuristic algorithms.

	Solution quality			Cost			Cost per improvement		
	Best fitness	Competi-tiveness	Equal population	Time (in seconds)	Iterations	Iterations per second	Improve-ments	Iterations	Time
Simulated Annealing	0.1237	0.0696	0.3402	1,489.12	1,472	0.99	130	11.32	11.45
Greedy	0.0980	0.0659	0.2264	3,817.91	186,619	48.88	858	217.50	4.45
Tabu	0.0984	0.0658	0.2288	1,968.94	92,659	47.06	794	116.70	2.48
GRASP (default)	0.0980	0.0659	0.2264	3,910.87	186,619	47.72	858	217.50	4.56
GRASP (contiguous)	0.0411	0.0491	0.0093	19,140.91	320,386	16.74	2075	154.40	9.22
PEAR snapshot 1	0.0403	0.0425	0.0316	1,555.64	45,358	29.16	44	1030.86	35.36
PEAR snapshot 2	0.0326	0.0303	0.0419	3,511.02	99,300	28.28	65	1527.69	54.02
PEAR snapshot 3	0.0219	0.0199	0.0300	5,207.64	150,577	28.91	112	1344.44	46.50
PEAR snapshot 4	0.0180	0.0176	0.0198	6,746.70	200,653	29.74	121	1658.29	55.76
PEAR snapshot 5	0.0167	0.0139	0.0279	8,963.47	270,732	30.20	135	2005.42	66.40
PEAR final	0.0145	0.0117	0.0256	10,784.55	329,572	30.56	142	2320.93	75.95

McDonald, 2011). There are not many software choices that perform functional automated redistricting for realistic redistricting applications (Altman and McDonald, 2011). We choose to present a comparison with BARD v1.2.4 because it is a known R package that provides a set of heuristic algorithms for redistricting. BARD supports methods for generating initial redistricting plans, including random generation (Grofman, 1982), random but contiguous equi-populous districts (Cirincione, Darling and O'Rourke, 2000), and simple and weighted k-means based generation. In addition, it provides several heuristics to refine initial plans, including simulated annealing, greedy search, tabu search, and Greedy Randomized Adaptive Search Procedure (GRASP). BARD formulates objectives and constraints as score functions that can be defined by users. For computational efficiency, it employs efficient data structures and native C libraries for routines which exhibit poor performance in R.

In the BARD implementation, the greedy algorithm is a hill-climbing local search method. The tabu search uses the greedy algorithm to explore the solution space but maintains a tabu list to avoid duplicating search efforts in similar solution space regions. The GRASP algorithm is a multi-start greedy algorithm with multiple randomly generated starting candidate solutions. Lastly, the simulated annealing algorithm conducts a probabilistic search of the solution space and then conducts another hill-climbing search after the annealing process has completed. All of the heuristics create new solutions by examining exchangeable units near district borders through 1- or 2-exchange operation.

When we ran these heuristics, we used a seed solution to begin the search process. Contiguity is, of course, legally required in the output plan. It is worth noting that if the seed solution was not contiguous, BARD exhibited great difficulty in producing contiguous plans during the search. Likely, given the problem size (2690 units), the connectivity graph is too large to allow a fast merge of any two disconnected subsets of a same district. To bypass this issue, we generated only randomized contiguous plans to seed the BARD heuristics. In addition, because BARD's implementation of the area–perimeter compactness is sufficiently inefficient to make the computation prohibitively slow, we did not include compactness in the specification of the objective function. Instead, the fitness function in this experiment is a weighted sum of the competitiveness and equal population measures. We also ran two versions of GRASP. The default GRASP setting uses random sampling, which generated numerous violations of the contiguity requirement and hence was unable to produce comparable results. To fix this issue, we modified the R code for GRASP to constrain it to use random contiguous samples. This run is reported as "GRASP (contiguous)."

Table 3.2 displays the results from the five BARD runs and one sequential PEAR run. The best fitness value, the computation time, and the number of iterations required to achieve the best fitness are measured. Each heuristic ran until either the search converged or the stopping criteria were met. Both GRASP versions experienced long starving times (over 19 hours) without improvement after the reported best fitness values were found. We had to terminate both runs manually. Among the five BARD runs, GRASP (contiguous) produced the best fitness (0.0411), but it took more than 5 hours to reach this fitness and was not able to improve upon this fitness even after an additional 19 hours of subsequent searching effort. The simulated annealing algorithm produced the solution with the worst fitness (0.1237), though it took the fewest number of iterations. Its iterations were the slowest, taking about a second for each iteration to complete. The greedy and GRASP (default) heuristics had almost identical performance: while their solution quality was the second best, they ran more quickly (48.88 and 47.72 iterations/second compared to 0.99 iterations/second for simulated annealing). The solution found by the tabu search is slightly worse than the greedy and the GRASP (default) solutions, and took much less time to compute, likely because of the use of a tabu list. It, however, suffered from early convergence.

We ran the sequential PEAR algorithm for three hours. The sequential PEAR run handily outperformed all five BARD runs, obtaining better solution fitness and doing so in much less time. In Table 3.2, we display snapshot results (every half hour) of PEAR's metrics beginning after 1,555.64 seconds into the run, at which time the sequential PEAR run identified a solution with fitness value 0.0403, which is better than the best result found by any of the BARD runs. As we can see from the snapshots, PEAR continued to steadily improve its solution quality without early termination or a long starving period during which no better solution is identified. The best fitness obtained by the sequential PEAR run is 0.0145, which is significantly better than any of the solutions produced by any of the BARD heuristics.

Table 3.2 also presents metrics for assessing the numerical efficiency as measured by the cost to gain fitness improvement. We present the overall computation cost as well as the computation cost per improvement. Since the iteration speed for the simulated annealing algorithm was so much slower than the other algorithms, its metrics are accordingly affected. For the BARD heuristics, the tabu search improved the most quickly, finding a better solution every 2.48 seconds. Neither the greedy nor the default GRASP algorithms was able to identify significant improvements. While they made a number of improvements (858), each improvement was sufficiently small that they did not make a significant contribution to the solution fitness. More pointedly, the total number of search iterations (186,619) eclipses this

relatively small number of modest improvements. The GRASP (contiguous) run benefited from the multi-start strategy, but took almost 5 hours more than PEAR to obtain its best solution. In comparison, the sequential PEAR algorithm initially appears to be slow in identifying improvements. On average, it took 1030.86 iterations and 35.36 seconds to make each improvement in the first snapshot. However, these improvements were substantial; the spatial EA operators were able to make significant gains in solution quality with a relatively small number of improvements. This performance can be attributed to two specific aspects of the PEAR algorithm. First, unlike the BARD heuristics that concentrate on improving a single solution, the evolutionary process in the sequential PEAR heuristic attempts to improve an entire population of solutions. The greater population size exacts a greater cost in iteration terms. Second, PEAR's more dramatic improvements clearly indicate that its numerical efficiency advantage has its roots in the utility of the evolutionary algorithm and the effectiveness of its spatial EA operators, not from the sheer quantity of iterations per second. Though it became increasingly difficult to find better solutions under tighter fitness thresholds, as evidenced by the increased cost per improvement shown in the subsequent snapshots, the sequential PEAR nonetheless steadily progressed toward better solutions.

In summary, the EA in PEAR was able to find significantly better solutions overall, was more effective and efficient in its search, and steadily improved even at increasingly tighter fitness levels in these experiments. We attribute the performance differential to our consideration of the redistricting problem as an inherently spatial optimization problem, and thus to our design of EA operators that explicitly, effectively, and efficiently incorporate spatial configurations.

3.6.3 Scalability Analysis

With an effective sequential EA as the base, we now evaluate the capability of our parallel algorithm to scale its performance with the number of processors employed. Strong scaling tests, which increase the number of processors while keeping the global population size intact, are problematic for measuring the performance of a PGA/PEA because the time taken to reach a solution quality threshold depends on both the amount of numerical work done by all of the processors and the stochastic nature of the EA search as demonstrated by (Liu and Wang, 2015). Instead, we evaluate the weak scaling of PEAR. In PEAR weak scaling tests, "data size" amounts to the numerical work done by each process instead of the number of problem variables (VTDs). This definition of weak scaling is termed "new-era" weak scaling (Sarkar, Harrod and Snavely, 2009). Given that Liu and Wang (2015) has

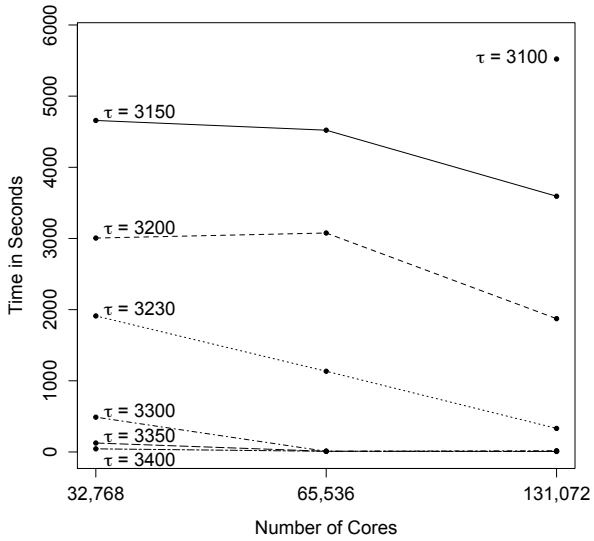


Figure 3.6: PEAR weak scaling test results at varying solution quality (fitness) thresholds, τ.

demonstrated the advantages of asynchronous migration over synchronous migration, we use only asynchronous migration for our PEA. In our experiment, the population size of each process is set to 200. The size of the global population across all the processes is thus 200 times the number of processes. To avoid precision issues for floating point number comparison on fitness, the fitness value of each solution is multiplied by 10,000. The fitness function is a weighted sum of competitiveness, population deviation, and compactness, as specified in Table 3.1. Because our EA operators do not inherently consider compactness, including compactness as a weighted component in the fitness function makes fitness improvements highly dependent on compactness, allowing us to observe the gains of utilizing more parallel computing power in our parallel algorithm that is separate from the efficiency afforded by our spatially cognizant EA operators.

Figure 2.14 shows the results of the weak scaling test from a two-hour PEAR run using 32,768, 65,536, and 131,072 integer cores on Blue Waters. The time taken to reach multiple solution quality thresholds (normalized fitness values; smaller value indicates better solution) is measured. In the plot, the solution threshold values are indicated besides each line/point. The downward sloping lines illustrate that for each solution quality threshold, utilizing more processors generally reduced the amount of time required to reach each threshold. In addition, as evidenced by the single point in the upper right of the plot, we were only able to reach the tightest threshold of 3,100 when we utilized 131,072 processors. Together, these

results indicate that our algorithm scales well with desirable outcomes as the number of processors increases.

In total, 1,174,702 unique feasible solutions were generated (167,719 from the run using 32,768 cores; 337,165 from the run using 65,536 cores; and 669,818 from the run using 131,072 cores). The number of feasible solutions increases fairly linearly with the number of processor cores. The best solution found using 32,768 cores has a fitness value of 3,133.98. The best solution found using 65,536 and 131,072 cores improved this value to 3,125.88 (8.10 less) and 3,083.16 (50.82 less), respectively. It is well known that as the solution fitness approaches the optimal, it becomes much more difficult to find better solutions. Accordingly, the benefit of solution fitness improvement in this study demonstrates the direct benefit of harnessing more computing power to enhance the problem solving capabilities of an EA solver.

3.6.4 Large Set of Feasible Solutions for Statistical Analysis

Solutions obtained from the scalability analysis are used for redistricting analysis. In cases of partisan gerrymandering, the Court has made it clear that it has the judicial obligation to determine "how much unfairness is too much" (126 S.Ct. at 2638 (n. 9)). To fulfill this responsibility, the Court needs data and measures to assess redistricting maps. Our algorithm generates a large number of good solutions, *two orders of magnitude more* and more efficiently than any other existing algorithms. Having a large number of solutions is essential for understanding redistricting maps because it allows us to place any particular map into context. Others have attempted to simulate maps (Cirincione, Darling and O'Rourke, 2000; Chen and Rodden, 2013), though none of those attempts even remotely approached the quantity or quality of the maps created by PEAR. We present two substantive analysis based on redistricting data from North Carolina and Maryland.

Redistricting in the state of Maryland. We now turn to an illustration of how our method sheds insight into the partisan gerrymandering debate with data from Maryland. An important consideration here is how one might construct a counterfactual for analysis. We have an electoral map that satisfies a set of legal and non-partisan criteria. This always includes population equality and may include, depending on the jurisdiction, criteria such as, for example, compactness and respect for political subdivisions. We know that the line drawers sought to satisfy some set of non-partisan criteria but are unsure whether or to what extent partisan considerations may have also come into play. Given this scenario, *the counterfactual set of maps is the set of plans that are at least as good or better on non-partisan factors because these are known considerations, but do not consider partisanship.* If the line

drawers did not consider partisanship, then their plan should resemble the broad contours of this counterfactual set. The important gain from the computation is that while it is difficult to objectively ascertain how much a human considered partisanship in the drawing of lines, it is not difficult to ensure that partisanship is not a consideration in a computer-drawn map because it is simple to either specify and/or verify that partisan data not be used at all or to set the level at which partisanship will be considered in relation to other non-partisan factors. For a partisan gerrymandering case, having a set of plans that are drawn without partisan considerations but exhibit comparable non-partisan metrics allows us to see how the alleged partisan considerations in the disputed plan substantively alter the outcomes that emerge from a less or non-partisan process.

To engage in creating this baseline reasonably imperfect comparison set, we begin by taking measurements from the disputed plan. For Maryland, on two non-partisan measures, population equality and compactness, we required that a reasonably imperfect plan have non-partisan metrics that are at least as good as the disputed plan. We deem these "goodness thresholds." Certainly, additional non-partisan criteria may be included here. These non-partisan factors should be chosen to match the peculiarities of the legal case. In our algorithm, as long as a particular facet can be quantified, we can incorporate it into our algorithm. We did not incorporate any partisan information into our map-drawing process.

For the state of Maryland, our algorithm identified more than a billion "legal maps." Of this set, about 250 million of these maps satisfy the requirements of reasonably good and feasible maps that would fit in the baseline set (i.e. they are as good as the current map on non-partisan criteria). From the 250 million maps, we retained only one feasible map every time 1,000 maps were generated. This is an additional way to inject independence among the set of maps, leaving us 258,584 maps for our final comparison set.

Responsiveness is a measure of how sensitive seat gains are to changes in vote proportion. The plot on the left in Figure 3.7 shows how the responsiveness of the current electoral map compares with the responsiveness of the 258,584 maps that were retained as our baseline comparison set. As we can see from the plot, in Maryland, there are many maps that are legally viable, at least as good as the current map on chosen non-partisan indicators, but are yet more responsive to the vote. In fact, of the set of reasonably imperfect maps our algorithm identified without using any partisan data whatsoever, 94.79% of the generated maps were more responsive to changes in the vote proportion than the current map. This implies that partisan considerations were likely at play in devising the current map since creating a map

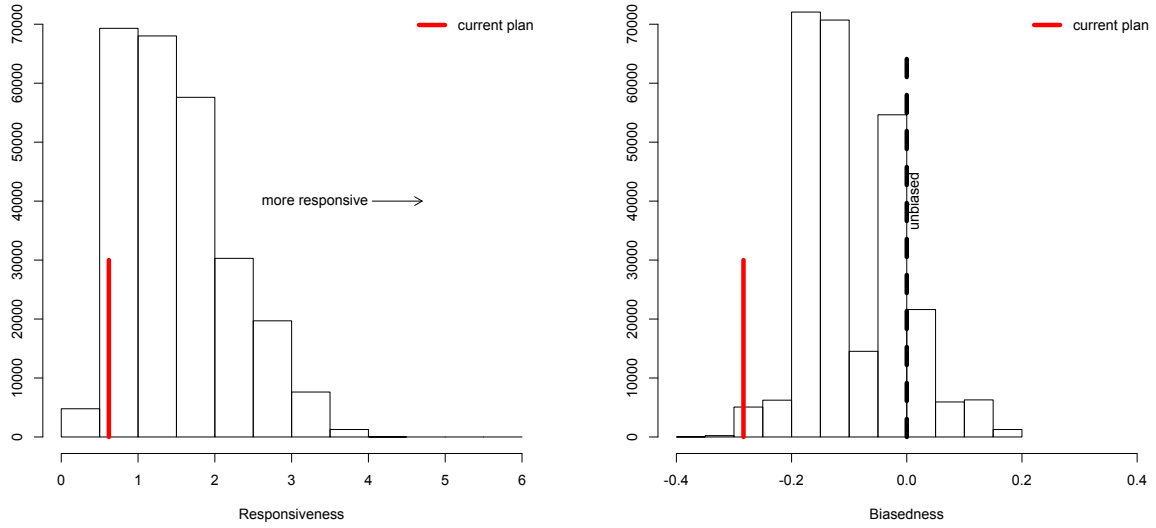


Figure 3.7: Current Plan versus 258,584 reasonably imperfect plans on responsiveness and bias.

with its level of responsiveness is unusual (though possible) when partisanship is not a factor in the map creation.

Biasedness is the condition of favoring one party over the other and can be described as a deviation from bipartisan symmetry. In other words, both parties should expect to receive the same number of seats given the same vote proportion. Here, the ideal symmetry is achieved at 0 while values that are negative show a bias toward the Democrats and positive values indicate a skew toward the Republicans. Measures of biasedness are shown in the plot on the right in Figure 3.7. Like the responsiveness plot to its left, the biasedness of the current Maryland map falls on the left end of the histogram. The current map is quite skewed in its favor toward the Democrats. Among the maps in the baseline set, the current map is more extreme than 99.79% of the generated maps on the biasedness measure. In Maryland, it appears that while the population landscape tends toward maps that seem to virtually always favor Democrats, the current plan has about as strong a Democratic bias as any plan we could identify with comparable non-partisan characteristics, again implying that partisan information was used in building the current map.

There are also other ways to measure the influence of partisan motivations. For instance, one might imagine that the most competitive (and therefore non-partisan) map would have districts that are evenly divided between Democratic and Republican registrants, and that

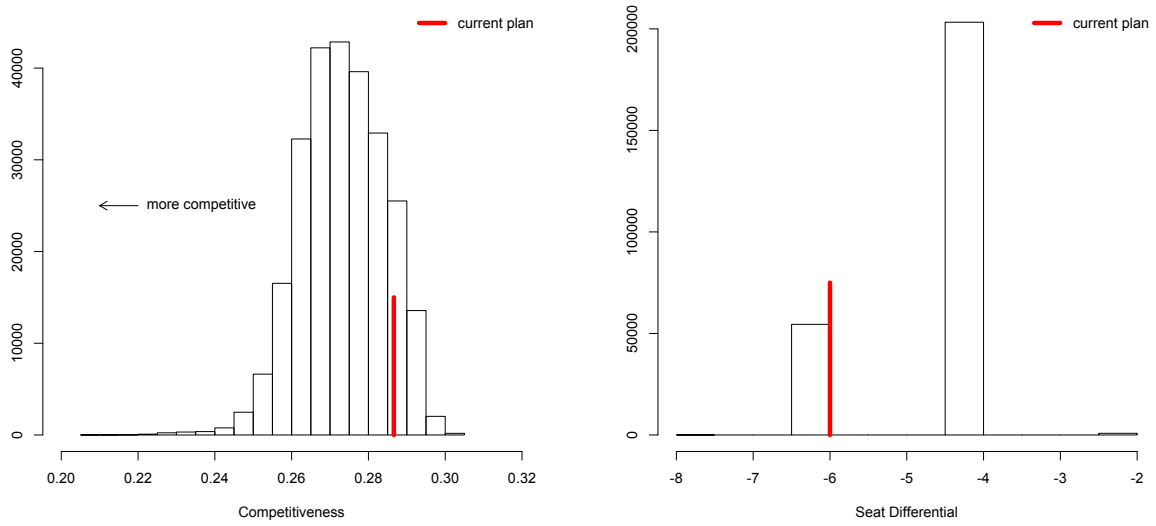


Figure 3.8: Current plan versus 258,584 reasonably imperfect plans on competitiveness and seat differential

any advantage toward one party or the other would be symmetrically split. That is, if there are eight districts, then if four districts leaned slightly Democratic, then the other four would lean slightly Republican. One way to quantify this measure of competitiveness is through the formulation in Equation 3.3.

The histogram on the left in Figure 3.8 shows how the electoral maps fall on competitiveness. It appears that the geographic landscape and the constraints on compactness and population equality constrain competitiveness to some extent. However, without considering partisanship, the large proportion of our maps were more competitive than the current Maryland map, which was more competitive than only 12.44% of our generated maps. It appears likely, then, that the line drawers used information that results in the maps being less competitive than they would be if no partisan information was considered.

In the right plot in Figure 3.8, we see the seat differential between the Democrats and the Republicans. In the vast majority (78.61%) of our generated maps, the Democrats have a 4 seats advantage. That is, the Democrats have an advantage in 6 seats while the Republicans have an advantage in 2 seats. In a smaller set (21.07%), they have a 6 seats advantage (as they do in the current map with 7 Democratic seats and 1 Republican seat). In a few plans (824 or 0.32%), the generated map is as good as the current map on non-partisan factors and has only a 2 seat differential (or Democrats with 5 seats and Republicans with 3 seats).

In 8 plans ($< 0.0002\%$), the Democrats have the advantage in all 8 seats while the map is as good as the current map on non-partisan factors, even though no partisanship information was used in the construction of the maps.

Since partisanship was not explicitly considered in the creation of our maps, we can regard the set of maps that we create as representative of the types of maps that are generated with no partisan intent but are still constrained by Maryland's natural population landscape, e.g., where the rivers and ocean flow, how the mountains carve up the state, how the cities have developed, the shape of its counties, the racial and/or socio-economic concentrations that have formed over the course of the state's history, and constraints on population equality and compactness. Some of these maps will appear highly partisan, but our concern is not the extremes of our distribution. Instead, we look to the first two moments of our distribution, the mean and the variance, to quantify the levels of partisan effects that are not excessive for the landscape under consideration. Accordingly, when we notice these levels of partisan effects that are identified by our analysis, the Court should not regard them as excessive or in need of regulation. On the other hand, if the disputed plan registers partisan levels at the extremes (where extreme defined by the Court) of the baseline set, then there may be cause for concern.

The distributions allow us to engage in a statistical analysis to assess whether the existing plans are outliers among other reasonably imperfect plans that could have been drawn, allowing us to understand whether, among the possibilities, is this proposed plan particularly unresponsive to voter preferences? Is this proposed plan exceptionally biased toward one party? Could we have achieved the goals of this new plan while maintaining greater respect for other important criteria or traditional districting principles such as respect for political subdivisions or compactness? Is the shift toward a Republican or Democratic bias a function of shifting demographics and population migration or are the motivations of the partisan line drawers the driving force? If the proposed plan is not exceptional in any way but is still biased toward one party, then the Court may decide that the grounds do not exist for revisiting the proposed plan. The pivot lies not within the plan itself or simulations based on one particular plan, but in how that plan compares to other possibilities. In this way, the ability to generate and analyze a large number of feasible redistricting plans without making a host of simplifications for computational reasons is essential for ensuring our democratic values.

Our plans still exhibit partisan effects as all maps do, but inarguably, our maps are not *intentionally* biased and have no underlying partisan motivation. Drawing maps in this way

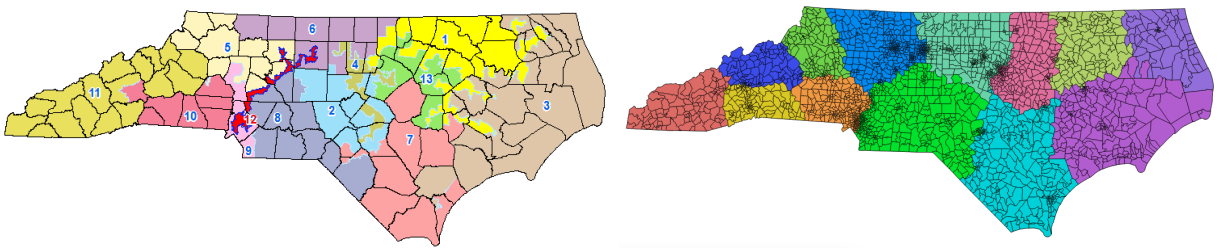


Figure 3.9: The 2011 Rucho-Lewis plan and one initial compact plan

allows us to separate how the population patterns in the state constrain the map making and how the partisan motivations might alter the creation of maps. If partisanship is not considered at all, it appears that the modal plan is more responsive, less biased, more competitive, and would give Democrats an advantage in 6 seats and the Republicans an advantage in 2 seats. In terms of responsiveness, the map under contention is particularly unresponsive to voters. It is also particularly biased. These histograms provide evidence that the map has encroached one party in favor of the other, and that these infringements were the result of an explicit consideration of party, not necessitated by the population landscape. Whether this impingement is excessive is left to the Court. Our analysis also shows that there are a slew of map alternatives that would significantly remedy the disputed plan's partisan effects while maintaining respect for the non-partisan criteria. Our analysis here is illustrative of what can be achieved. In an actual legal case, the lawyers or the judge may wish to expand the non-partisan constraints to include, for instance, respect for particular political subdivisions. We did not include that here, but this is a simple extension. Each legal case is idiosyncratic. Our model is easily pliable to these type of particulars.

Redistricting in the state of North Carolina. We present a similar analysis for North Carolina. For this analysis, the existing 2011 Rucho-Lewis plan (shown on the left in Figure 3.9) was used as a seed solution for PEAR. The figure on the right in Figure 3.9 shows one of the initial compact plans that is generated with our population initialization strategy. Our algorithm was able to create more than 1 million reasonably good feasible solutions satisfying the defined competitiveness, population deviation, and compactness principles.

Figure 3.10 summarizes the 1,174,702 solutions identified by our algorithm on three particular criteria in redistricting research: competitiveness, responsiveness, and biasedness. The green line shows how the 2001 map fared on each criteria while the red line indicates how the current 2011 map compares. The distributions allows us to assess whether the existing plans are outliers among other plans that could have been drawn. We can then understand

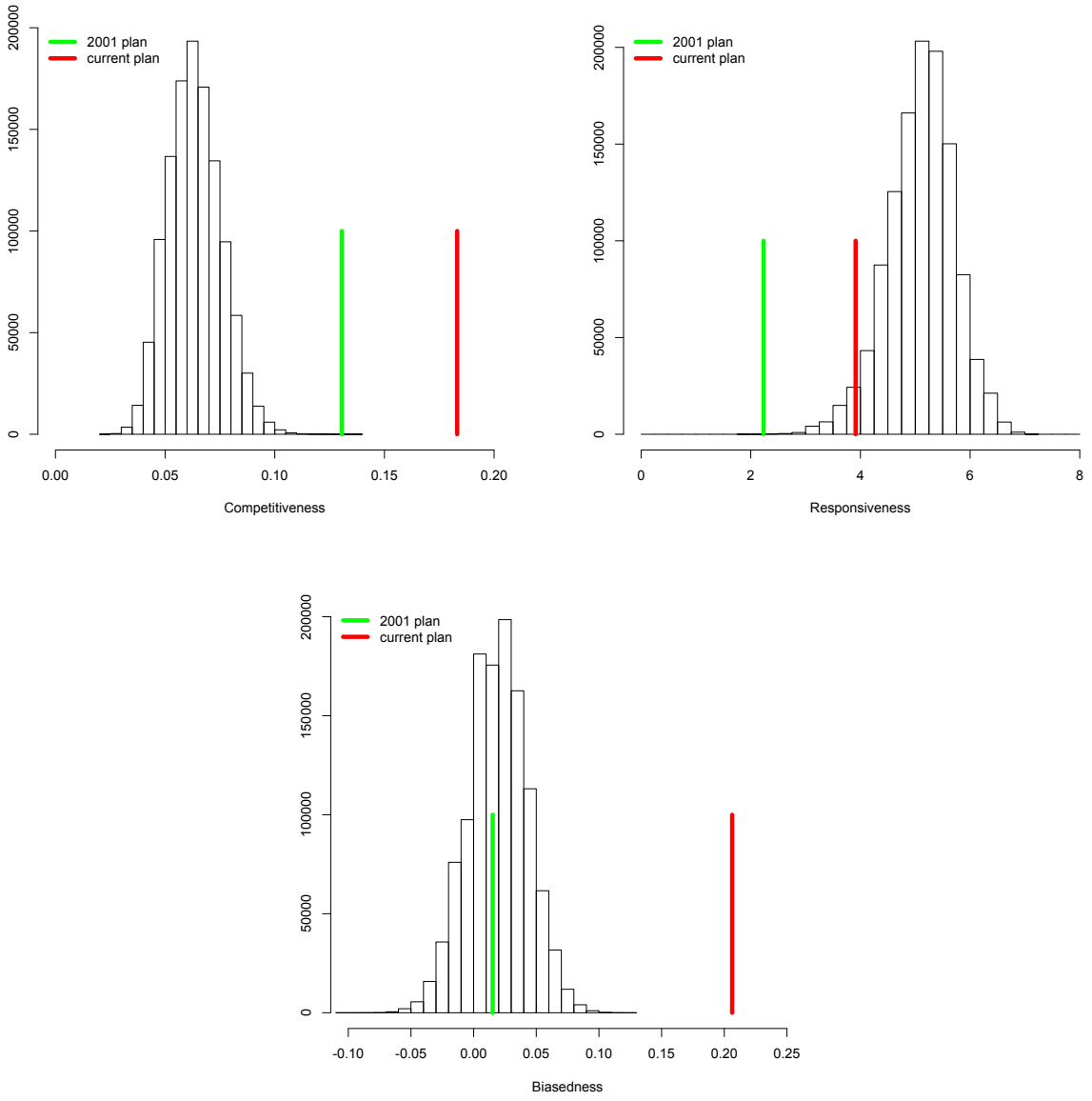


Figure 3.10: Current plan versus 1,174,702 reasonably imperfect plans on multiple redistricting criteria

whether among the possibilities, is this proposed plan, for instance, particularly unresponsive to voter preferences or exceptionally biased toward one party? Could we have achieved the goals of this new plan while maintaining greater respect for other important criteria or traditional districting principles such as respect for political subdivisions or compactness?

In the Maryland analysis, we demonstrated how one might examine the question of partisan gerrymandering. However, our tool is malleable to an array of different questions about representation. Importantly, the pivot to gaining insight into these questions lies not within the plan itself or model simulations based on one particular plan, but in how that plan compares to other possibilities. Without the ability to generate a range of plans, we would not be able to make reasoned arguments about the impact of redistricting on the electoral process. We would, moreover, not be able to persuasively make the types of legal arguments that are required by Supreme Court mandates. Plainly, the ability to generate and analyze a large number of feasible redistricting plans is essential for ensuring our democratic values.

Statistical approaches with limited data have been tremendously insightful on any number of realms. More recently, with the proliferation of significant computing power, we have discovered the extensive and often surprising reach of technology, information, and computation into many realms of life. These very same capacities can shed insight into our governance structures, ideally enabling us to improve our democratic society. This is our approach here—to integrate technological advances with our articulated strategy for analyzing, contextualizing, and understanding redistricting plans.

The theoretical underpinnings of statistical models also highlights challenges for our solution space traversal. The usual search for an optimal solution must be modified to a traversal with the purpose of yielding a set of high quality, independent solutions. The redistricting applications pushes us to adapt our heuristics for statistical frameworks with astronomical solution spaces where the independence requirement constitutes a significant additional challenge for standard optimization methodologies.

The recent proliferation of significant computing power has helped us discover the extensive and often surprising reach of technology, information, and computation into many realms of life. These very same capacities can shed insight into our governance structures, ideally enabling us to improve our democratic society. This is our approach here—to integrate technological advances with our articulated strategy for analyzing, contextualizing, and understanding redistricting plans.

3.7 Conclusion and Discussion

Our computational approach is designed to identify redistricting maps that satisfy a set of user-defined criteria with a particular focus on addressing fine levels of spatial granularity. We leveraged and enhanced a scalable PGA library to develop PEAR for the computationally intensive redistricting problem. By incorporating a set of spatial configuration operators and spatial EA operators to handle spatial characteristics and the associated computational challenges in the EA search process, and harnessing massive computing power provided on supercomputers, PEAR provides a powerful and computationally scalable redistricting tool.

While we have made considerable progress, numerical efficiency can be and needs to be further enhanced before our redistricting tool can become fully functional and practical for general application. Generating a large number of maps, which we have now achieved, is important and fundamental for the subsequent statistical analysis. Indeed, while our work has yielded significant gains, the computational complexity of the redistricting problem is formidable, leaving much progress to still obtain.

The major limitation of the current implementation stems from the design of crossover and mutation operators in PEAR. These operators are modified from classic linear recombination methods in order to consider spatial properties such as contiguity, hole-free, and non-overlapping partitioning of districts. While the mutation operator exhibited desirable performance, the crossover operator performed poorly because the overlap operation disrupts spatial attributes in parents and makes it very hard for the repair operation to construct a new feasible solution from the splits. Similar approach employed in literature Xiao (2008) showed promising results for small problems. But for the redistricting problem we tackled, the crossover operator needs to be much more effective. In the next chapter, we study this limitation further and propose a new EA framework for spatial recombination.

CHAPTER 4

SPATIALLY EXPLICIT EVOLUTIONARY COMPUTATION

Spatial optimization (SO) methods seek to allocate or arrange spatial units in a way that optimizes some objective or measure of goodness. Optimally arranging spatial units has many applications, and spatial optimization problems have been explored in a wide variety of contexts, dating easily back to the 1800s with explorations of efficient land use activity (Von Thunen, 1842). More recent research includes management plans for the national forests (Kent et al., 1991), studies of metabolic regulation (Agapakis, Boyle and Silver, 2012), location models (Church, 1999; Craig, Ghosh and McLafferty, 1984; McLafferty and Ghosh, 1986), coverage problems (Farahani et al., 2012; Murray, 2016; Murray, O'Kelly and Church, 2008), and land use allocation problems (Ligmann-Zielinska, Church and Jankowski, 2008). Across an array of different areas, research in spatial optimization is lively and generates significant interest. Many spatial optimization problems are computationally challenging and *NP*-Hard (Garey and Johnson, 1979). Examples include the regionalization problem that aims to group spatial units into a few regions that satisfy optimization objectives, e.g., p-region problem (Duque, Church and Middleton, 2011), spatial clustering-based regionalization (Guo, 2008; Wang, Guo and McLafferty, 2012)), the redistricting problem that seeks to partition spatial units into a set of contiguous districts/zones (Liu, Cho and Wang, 2016), the maximal covering location problem where the coverage of a set of facilities is maximized (Francis and White, 1974; Church and ReVelle, 1974; Schilling, Jayaraman and Barkhi, 1993; Farahani et al., 2012), the spatial allocation problem seeks to allocate specific activities to defined spatial units (Hof and Bevers, 1998), and the p-Hub location problem that locates p transportation hubs and allocates demand to specific hubs so that total transportation costs are minimized (O'Kelly and Miller, 1994).

Spatial optimization problems pose interesting and unique challenges for traditional optimization methods because the heuristics must intelligently incorporate spatial information. Failure to do so often leads to ineffective and inefficient heuristic algorithms, which is especially problematic for computationally complex applications. Indeed, as the size and richness of spatial data have increased, the associated spatial optimization problems have

become progressively more intricate, characterized by massive decision spaces that eclipse the capabilities of exact algorithms to identify optimal solutions. Innovation in spatial optimization methods is essential for progress and continued success in deploying future spatial optimization applications.

One path for solving difficult search and optimization problems is via Evolutionary Algorithms (EA), heuristic methods inspired by natural selection (Holland, 1992). EAs mimic evolutionary processes with a set of solutions encoded into a population at the initialization stage. Through stochastic EA operators (e.g., selection, crossover, mutation, and replacement), the population evolves based on a "survival of the fittest" rule (Wright, 1932; Goldberg, 1989). This route has been successful for many large-scale optimization problems. Utilizing EAs for spatial applications, however, is not straightforward as one cannot simply apply an EA from existing math and GIS software or libraries. Instead, carefully designing spatial optimization methods (i.e., data structures, geometrically-aware operators, and spatially cognizant movement) is critical for performance. This is the task we undertake here—designing a generalized framework for spatial optimization within the EA rubric. We focus on large scale spatial optimization problems where resolving computational issues is especially critical. According to Xiao (2008), these problems can be categorized as either assignment or partition problems, with or without spatial constraints. We focus on partitioning problems with spatial constraints, which embed strong spatial constraints, and are considered to be the most challenging.

Although randomization and stochastic components powerfully enable EA performance, these fundamental components are sub-optimal for spatial applications without significant re-design. At issue is that the traditional EA encoding of the problem into binary strings is devoid of spatial information. New solutions, generated from flipping or exchanging bits on binary strings, easily violate spatial constraints (Liu, Cho and Wang, 2015). While it is possible, it would be highly unusual that a spatial optimization problem could be solved using only binary operators. A spatial evolutionary algorithms could rely on classic linear recombination operators (such as crossover and mutation) to generate new solutions, but then fix broken spatial relationships in new solutions as a post-processing step. We find this strategy to be computationally expensive and instead argue that *EA recombination operators must be not only spatially aware, but also spatially explicit in exploring the spatially constrained decision space*. Such constructive methods are more promising than repair methods, particularly for large problem sizes where reducing computational cost requires consideration of an enormous decision space that is characterized by a large number of infeasible solution

areas. It is always good practice to avoid computation that results in infeasible solutions, and while repair operators can be effective, if the number of infeasible solutions is large, the repair operators exact an increasingly high computational cost.

We begin with a literature review in Section 4.1 that highlights how research in geographic analysis has approached spatial optimization. In Section 4.2, we present our methodological strategy that bolsters efficiency and effectiveness by avoiding infeasible solutions through constructive spatial neighborhood functions and graph-aware operations that access and maintain data structures that store spatial properties and relationships. In Section 4.3, we present our algorithm, a generalization of the path relinking framework adapted with spatial operators that slot well into the foundational theory underlying evolutionary algorithms by retaining the benefits of the randomization and stochastic elements that are central to EAs. In Section 4.4, we evaluate our operationalization with a redistricting application. Finally, we provide some discussion and conclude in Section 4.5.

4.1 Literature Review

Adapting heuristics to function efficiently with spatial data has been a consistent effort in geographic analysis research. Openshaw and Rao (1995), in developing tabu and simulated annealing heuristic methods for census zoning, first introduced a contiguity-preserving method that generates new solutions by morphing only on zone boundaries. Tong, Murray and Xiao (2009) implemented a spatial GA for the maximum coverage problem with a specialized crossover procedure that avoids spatial clusters and promotes spatial dispersion. Xiao, Bennett and Armstrong (2002) designed a spatial multi-objective EA for the site search problem by modifying both the mutation and crossover operators to take spatial considerations into account. Location-based operators were devised to move a subset of units to a location randomly identified or in another solution while morph-based operators identify moveable units between two spatially correlated (e.g., contiguous) sites and exchanged their site assignment (thus "morphing" the shape of sites). Xiao (2008) provides a further generalization with a graph theoretic approach for categorizing spatial optimization problems into partitioning and assignment problems. Wei and Murray (2013) designed a multi-objective EA for optimizing spatial uncertainty by treating uncertainty as an objective and constraint. Wu, Murray and Xiao (2011) designed a multi-objective EA for optimizing spatial contiguity in a reserve network design by incorporating relative contiguity (Wu and Murray, 2008). Cai, McKinney and Lasdon (2001*b*); Wang (2013) conducted water resource modeling using genetic algorithm for optimization for water resource management. Cai, McKinney and

Lasdon (2001*b*) applied a GA+LP approach to study two nonlinear spatial and hydraulic models: a reservoir operation model and a long-term dynamic river basin planning model. LP is used to calculate the penalty function. Wang (2013) developed a spatial EA for large ground water modeling. Spatial characteristics are encoded in an adaptive quad-tree data structure that provides the branches of the tree that are grafted from one parent solution to another in the crossover operation while splitting, merging, and alternation are applied on the tree in the mutation operator. Despite many efforts, there remain many interesting and challenging research directions for spatial optimization.

There are different considerations and distinct paths one may pursue to enable greater efficiency and effectiveness of spatial optimization searches. As an overview, Tong and Murray (2012) summarize relevant spatial elements in optimization problem formulation and solving mechanisms. Related to specific optimization algorithm design, the representation and handling of objectives and constraints capture some set of spatial characteristics, including distance, adjacency, contiguity, intersection, containment, shape, partitions, and pattern. Tong and Murray (2012) further highlight that spatial characteristics may be incorporated in the specification of decision variables, coefficients, functions, and constraints. intersection, shape, districts, and areal patterning.

Consider the various ways in which one might handle the spatial contiguity constraint. Here, usually one of three strategies is employed: contiguity is encouraged in the search process (Wright, ReVelle and Cohon, 1983; Minor and Jacobs, 1994); solutions are generated without spatial considerations but non-contiguous solutions are discarded (Brookes, 1997; Nemhauser and Woolsey, 1988), or contiguity is enforced by creating only contiguous aggregations. To accommodate contiguity as a strong constraint, contiguity must be part of the algorithm design. Cova and Church (2000) devised a series of math inequalities using 1D and 2D hamming distances to represent the principle that no spatial unit can be chosen before a unit closer to a reference unit is chosen. This representation permits an integer programming system to make implicit spatial neighborhood moves via branching that are based on distance. In this way, contiguity is maintained while the choice of different paths from a cell to the reference site impacts the value of objective function and other constraints. Cleverly, the hamming distance is defined through a directional spatial relation between a cell and its neighbors toward the reference site. Using similar representation, Williams (2002) addressed the exact contiguity constraint for vector-based problem formulation. Shirabe (2005) then optimized the contiguity representation for integer programming in a way that reduces the search cost for linear solvers. Church and Cova (2000) applied these contiguity

formulations in an application of map evacuation risk. Murray, Grubesic and Wei (2014) conceptualized unit connectivity as a network and defines contiguity to be a classic network flow problem, in which a connected graph allows the selection of paths from source to sink. This representation is used in the Contiguous-Max-LLR model to solve the spatial clustering of input data with irregular shapes. While the aforementioned representations satisfy the contiguity constraint, it is unclear how the underlying LP/IP solvers perform for efficient search space pruning.

In addition to designing how best to represent the contiguity constraint, incorporating spatially aware modifications involves thoughtfully counterbalancing the additional computation that is introduced by these adaptations. For example, in their spatial scanner that searches a map for clusters, Izakian and Pedrycz (2012) designed a particle swarm optimization algorithm that checked the contiguity of *each* cluster when using the centroid to include regions. King et al. (2012) designed a specialized data structure, the geo-graph, to check for contiguity in zones. This resulted in improved performance though applying the check at each EA iteration imposed a non-trivial computational performance penalty. Spatial optimization designs alleviate these additional computational burden. Liu, Cho and Wang (2016) applied an inductive contiguity checking strategy that improves efficiency by forcing contiguity in initial solutions and checking for contiguity only in updates rather than in the entire solution. This strategy improved the efficiency of contiguity checking by two orders of magnitude.

Contiguity also plays an important role in developing heuristics for spatial clustering. Duczmal et al. (2007) developed a GA crossover operator to optimize a clustering on irregular shapes. The contiguity-preserving crossover overlaps two parenting clusters first. An overlapped connected component is then selected as a seed. A linear sequence is generated to include adjacent nodes of the seed gradually until all the nodes in a cluster are included. New contiguous clusters are generated by combining the two sequences from the two parenting clusters, respectively. For partitioning SO problems with spatial constraints, this spatially-aware crossover would be insufficient because it only considers the contiguity of one cluster while partitioning problems generate multiple contiguous districts.

To be sure, our goal is not simply to incorporate spatially aware operators into the heuristics, but to do so in a computationally feasible manner that will scale with problem size. In this pursuit, scalability can be fostered by avoiding spatially infeasible solutions, which can be accomplished either by designing operators that preserve the spatial constraint or by designing repair operators that restore the spatial constraint (Mezura-Montes and Coello,

2011). For the spatial partitioning optimization problem, Xiao (2008) designed an overlay and repair operator that overlaps two parent solutions and constructs a child solution from the subdivisions resulting from the overlap operation. Since the number of subdivisions is highly likely to be greater than the required number of partitions, k, a randomization algorithm is iteratively applied to select and merge two subdivisions until there are k subdivisions. This crossover operator is spatially aware but has two drawbacks. First, it is computationally inefficient. Two randomly selected subdivisions are not likely to be contiguous, so subdivisions must be chosen repeatedly until contiguous units are identified. Such trials are not efficient, especially when the problem size is large. Second, spatially overlapped subdivisions may significantly differ from their parents in both spatial characteristics, such as size and shape, and aggregated non-spatial attributes. For example, an overlapping may result in a few big partitions and a large number of scattered small partitions in partitioning problem-solving. In such scenario, the recombination computation may spend significant amount of computing time handling small subdivisions that contribute marginally to fitness improvement. Such side effect may lose the rationale of conducting crossover – inheriting subset of attributes contained in parent solutions.

While it is essential to incorporate spatial characteristics into the EA operators, we must be mindful that there is an inherent tradeoff between operator sophistication and algorithm convergence. Operators with complex logic may be spatially cognizant, but the complexity may adversely affect the time required for the algorithm to converge.

4.2 Methodology

Failure to recognize and incorporate spatial relationships and constraints significantly degrades the performance of EA operators in the decision space traversal. Conventional linear chromosome encoding fundamentally inhibits the ability to incorporate spatial information. This limitation can be overcome by modifying the EA operators to be spatially explicit operators. We incorporate theories underlying ejection chain and path relinking.

4.2.1 Maintaining Spatial Constraints in Decision Space Traversal

We begin with a simple illustration that shows how a contiguity requirement changes the decision and search space for the classic 1-bit EA mutation operator. For ease of illustration, our example employs a raster representation of the spatial variables. However, our algorithms applies to both vector- and raster-based problems since we adopt an adjacency

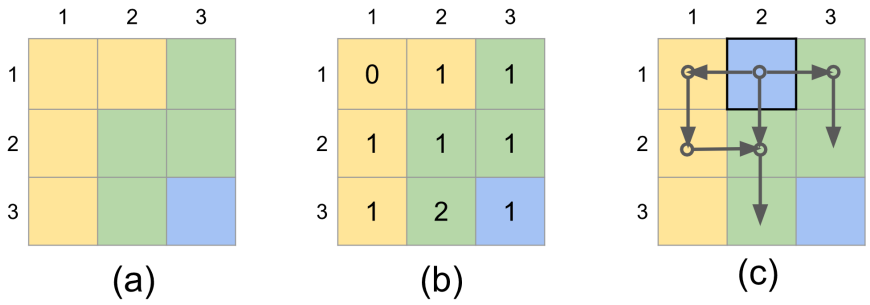


Figure 4.1: Search space of mutation for spatial optimization problems, if only rook neighborhood is allowed. (a) is a solution of 9 spatial variables on a 2D raster layout. (b) shows the number of possibilities each cell can have when chosen as the mutable cell. (c) illustrates search paths needed to repair a mutation result.

graph structure to represent neighborhood, which generalizes the adjacency of spatial objects of both regular (e.g., raster) and irregular shapes.

In Figure 4.1, the leftmost figure shows nine spatial units partitioned into three contiguous districts. Using a classic 1-bit mutation, the new solutions that result from mutating a single cell are shown in the middle figure. The number of possibilities is $2 \times 9 = 18$, because each cell can be re-assigned to either of the other two districts. However, note that mutating cell (1,1) breaks contiguity and necessitates a repair operator to restore contiguity. The rightmost figure in Figure 4.1 demonstrates a repair scenario. First, Cell (1, 2) switches from yellow to blue. The repair operator then connects it to the existing blue district $\{(3, 3)\}$ by identifying a route between them and re-assigning the cells on that route to the blue district while maintaining the two other districts. For example, the route $(1, 1) \rightarrow (2, 1) \rightarrow (3, 1) \rightarrow (3, 2)$ is not possible because it eliminates the yellow district. Similarly, the route $(2, 2) \rightarrow (2, 3)$ is not possible because it splits the green district. Repairing non-contiguous solutions is plainly computationally costly, even for a simple mutation operator. Thus, avoiding the need for repairs is desirable.

Part of our strategy to avoid repairs is to design appropriate data structures. This is necessary but not sufficient since we must also carefully design the operators on these data structures. To be sure, if spatial constraints are represented as non-linear data structures such as an adjacency graph, or even geometric structures, graph or geometric operations must still be implemented efficiently to attain desirable computational scalability. The main search complication that arises from imposing spatial constraints is that the search space becomes "patchy," where feasible and unfeasible patches of solutions are located next to one

another. The number of infeasible solutions is often enormous, so avoiding the patches of infeasible solutions dramatically increases efficiency.

4.2.2 Limitation of EA Recombination Operators

Classic EAs operators that linearly combine parent solutions have two main drawbacks. First, they limit the feasible search space and may miss feasible search regions, especially when spatial constraints make feasible regions difficult to find. Even in an unconstrained decision space, the search space derived from linear recombination is smaller than the enumerable possibilities (Glover, 1994). Second, since linear recombination is unaware of the non-linear decision space that characterizes spatial optimization, classic EAs often produces new solutions that violate spatial constraints, which either are discarded, leading to poor EA efficiency, or need repairing, a non-trivial and computationally expensive task.

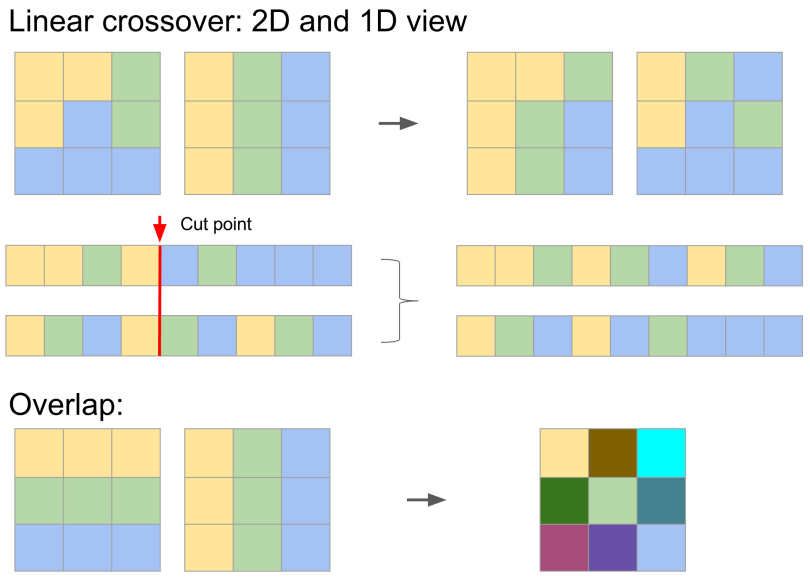


Figure 4.2: Limitations of classic EA crossover on spatial configuration. The upper diagrams illustrate a linear recombination crossover. The lower diagram illustrates an overlap-based crossover.

Figure 4.2 illustrates these issues with two commonly employed crossover operators. The first, the classic linear crossover using a single *cutpoint*, is shown in the upper diagram of Figure 4.2. Here, the two parent solutions are shown on the left as 2D and 1D encoded (chromosome) views, respectively. After a cutpoint is chosen, the crossover generates two new solutions (shown on the right) by swapping the second part of one chromosome and attaching it to the first part of the other chromosome. However, neither of the two new

solutions is contiguous. In this example, there are 8 cutpoints for 9 variables. Each cutpoint results in 2 new solutions. In total, there are at most 16 possible new solutions (including infeasible non-contiguous solutions, but not duplicate partitions).

The bottom diagram in Figure 4.2 illustrates an extreme case of a spatially-aware crossover operator proposed by Xiao (2008). This crossover operator overlaps two solutions and randomly organizes the resulting subdivisions into contiguous districts. A subdivision is identified with a unique <*district1*, *district2*> label which denotes a cell's district assignment from the two parent solutions. The goal of this type of crossover is to preserve the good attributes from both parent solutions. This is intuitively appealing, though its hope may not borne out as we can see from the subdivisions shown in the diagram. Notice that the extreme case of overlapping creates 9 subdivisions, which creates a new problem that is identical to the original problem rather than a starting point from which better solutions might be derived. More poignantly, since the overlap procedure depends on the graph cut in parent solutions, it is not clear when or how this procedure might generate better solutions that preserve and propagate "good" subdivisions. Despite these drawbacks, however, a significant benefit of the overlap crossover remains—it can create a search space of size at least $3^3 - 2 = 25$ (by taking the largest three subdivisions as starting districts and parceling out the remaining three cells, but subtracting the two existing solutions), which is a significant improvement over the classic EA crossover.

This discussion highlights that an important component has been missing from the extant EA *spatial recombination* operators. In particular, they do not consider spatial characteristics *during* the new solution generation phase. Spatial recombination operators must transform the spatial characteristics that are related to the optimization problem solving into quantifiable measures that direct the decision space search. This is akin to incorporating domain knowledge into embedded heuristics. The main difference is that spatial characteristics require an integrated framework for incorporating multiple spatial elements while maintaining the efficiency of each EA iteration. In this direction, Xiao (2008) made an initial effort by integrating graph theoretic components to categorize spatial optimization problems for EA design. This effort needs to be further extended, and more importantly, needs to be systematic.

4.2.3 Spatial Recombination Framework and Data Structures

For large spatial optimization, an efficient and effective spatial EA recombination operator must fulfill a number of requirements.

1. The representation of the decision space must *allow spatially explicit exploration* of the space. Since spatial and geometric relationships are often non-linear, data structures and the associated operations should also be non-linear.

2. The neighborhood function, which is the building block for exploring the decision space, must also be *spatial cognizant*. Defining neighborhoods in an unconstrained discrete decision space is insufficient for exploring and exploiting spatial configurations.

3. *Recombination strategies must be efficient*. Since the computational complexity of general graph traversal on n spatial variables may increase from $O(n)$ to $O(n \log n)$ for linear combination, inefficient search methods, such as those that rely solely on random trials, must be avoided.

4. *Recombination strategies must embody stochastic components*. While incorporating heuristic strategies for facilitating spatially explicit search, randomness remains critical for ensuring diversification in the search.

In short, we desire a **S**patially **E**xplicit **E**volutionary **C**omputation (SEEC) framework that adheres to the basic framework of EA recombination procedures, which seeks to pass desirable attributes from parent solutions to child solutions. This is accomplished within the design of a spatial search heuristic that is cognizant of spatial features and systematic in how those features are incorporated into the algorithm so that the computational burden is intelligently managed.

Our strategy is careful to consider the peculiar requirements of spatial recombination for large spatial optimization instances. In particular, for these applications, the solution space is often characterized by many long-ranging plateaus that rise and fall in the global space, but at any local point, the immediate local topological space is essentially flat. This occurs because moving any one spatial unit has minimal effect. Significant changes occur only with large moves. For this type of solution space, neighborhood functions that produce only local moves, whether these moves improve the solution fitness or not, will not lead to a significant change in solution fitness and are not particularly helpful in guiding the decision space search into more feasible areas where good solutions reside. Instead, such solution space distribution behooves recombination operators that are capable of generating large moves that can propel the search into new and promising search areas, but are also carefully orchestrated so that the underlying spatial configurations are not broken and that the desirable components of the parent solutions are retained and propagated.

Given that there are many objectives and considerations that must be accounted for, designing appropriate data structures is critically important. We embody the intentions of the algorithm into appropriate graph structures that incorporate spatial characteristics.

The spatial units can be the vertices while the edges can denote spatial connectedness (Xiao, 2008). Additional information can be incorporated via vertex and edge weights. Most spatial elements can be captured in this framework. For instance, the graph properties and graph search functions allow us to capture adjacency, contiguity, intersection, partition, route, graph distance, and patterns. For problems with graph representations, edge optimization routines like 2-opt or 3-opt for the Traveling Salesman Problem are often employed to make each search step small enough to control but flexible enough to explore new areas of the decision space. In addition, the search space can be subdivided into subgraphs through a combinatorial function that selects, splits, or merges graph vertices and edges. Spatial neighborhood functions can operate on subgraphs to make local moves, explore feasibility, or recover from infeasible moves. For example, a spatial neighborhood function that preserves adjacency and contiguity may search on the boundary of a subgraph to identify neighboring spatial variables adjacent to the subgraph. An exception to the flexibility of the graph representation is that there is no capacity to work with shapes, which requires the encoding of geometric information. However, from an optimization perspective, unlike binary linear recombinations which easily break spatial dependencies, the graph representation is flexible and its associated neighborhood topology structure permits exploration of interdependencies among variables and search regions that are imposed by spatial elements.

The graph structures also enable an easy way to create a chain of local moves that collectively comprise a large disturbance in the search neighborhood. Chained moves are desirable since a chain, which is tied on either end to feasible solutions, allows one to design globally large, but locally incremental moves, within the search space. The length of the chain may be adjusted to permit a controlled but sufficiently large move to make an impact. Auspiciously, chained moves can be seen as a generalization of the mutation and crossover operators in EAs. A mutation operator can be designed as a series of chained moves where the chain, at both ends, is anchored to a single parent solution. Similarly, a crossover operator can be designed to take two different parent solutions to connect the chain. A set of chained moves can then define a "walk" from one parent solution to the other parent solution. The movement along the chain generates a series of intermediate or child solutions. The length of the chain is defined by the distance (i.e., difference) between the two parent solutions and indicates the degrees of freedom for building a path. Coupling randomized local moves with incremental chaining provides a desirable constructive method for designing spatial recombination strategies that incorporate spatial neighborhoods and chaining on graphs. This observation inspires the design of our spatial recombination operators as spatial analogs and

adaptations of two well-established heuristic methods, path relinking (Glover, Laguna and Marti, 2000) and ejection chain (Yagiura, Ibaraki and Glover, 2004), both of which have successfully employed a chaining idea to efficiently and effectively solve large combinatorial (non-spatial) optimization problems.

4.2.4 Spatial Crossover through Path Relinking

A path relinking process begins by identifying two solutions, an initial (or source) solution, S, and a reference (or target) solution, T. The path between these two solutions is constructed and traversed with the hope that somewhere along the path (in the neighboring space) lie new and better solutions that are a mix of S and T. Designing how one traverses this path is a key component of path relinking methods. The path links multiple planned or random moves, where each move is an incremental change from the previous state. Path length is a function of solution distance from S to T. The distance for non-spatial problems can be defined as the number of variables with different values in the two parent solutions. At each step, a neighborhood function stochastically determines a local move. Each step forward, the distance is reduced along the relinking path and reaches zero when the path relinking is complete. Collectively, the moves transform one (initial) solution to the other (target) solution. These moves can be designed adaptively to improve the performance of the heuristic.

It is worth noting that EAs and path relinking are closely related. Glover (1994) relates GA and scatter search, a specific form of path relinking in linear programming. He highlights that the link between these two approaches is rooted in principles underlying mathematical relaxations. Scatter search, and its more general form, path relinking provide an ordered way to strategically explore the neighborhood space for generating new solutions. The neighborhood space is defined by the population, not the affinity of a single solution, and thus needs one or multiple parent solutions from the current population as input. In this way, scatter search exhibits commonalities with EAs since it is iterative, selects moves within a neighborhood, and performs recombination through the exploration of the neighborhood space.

These same ideas underlie the design of our spatial recombination methodology. We must, however, adapt existing path relinking and ejection chain methods away from non-spatial chromosome encoding. For example, solution distance, which is often defined as the number of variables with different values between two solutions, is not applicable in spatial configurations. Two spatial partition plans differ in the partition shape. Two zones

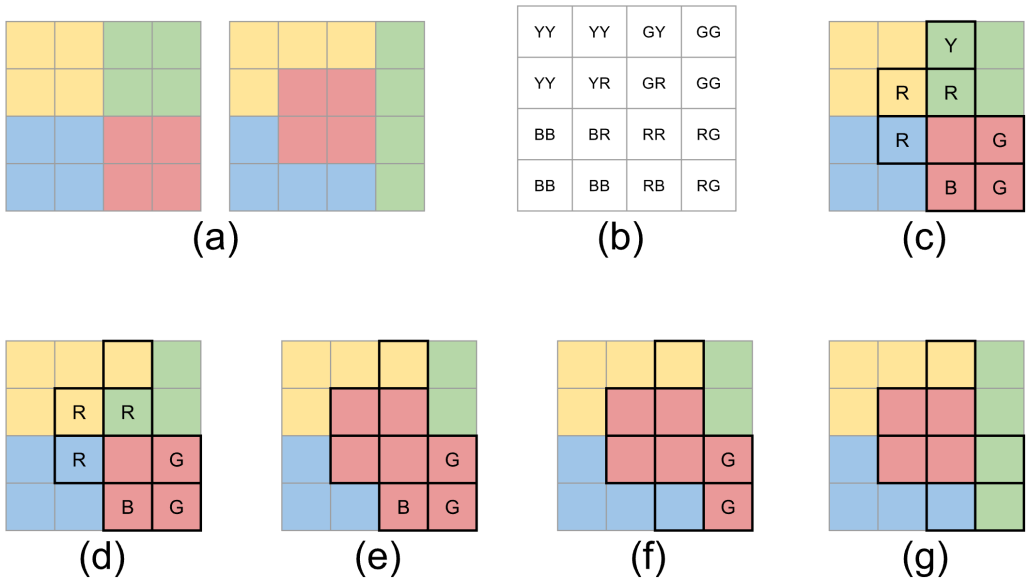


Figure 4.3: Spatial crossover PRCRX illustration. Letters denote colors, too.

of the same shape but different zone index are identical but would be considered different in a chromosome encoding. For this reason, while a path relinking process can surely walk from chromosome A to B by flipping bits that differ, it may not be able to walk from one spatial configuration to another because spatial constraints such as contiguity, containment, or shape, may shorten, lengthen, or even block the path. The design of spatial recombination operators is, thus, significantly different from classic crossover and mutation, as well as conventional path relinking and ejection chain methods.

We design two novel spatial EA operators, PRCRX (spatial crossover) and ECMUT (spatial mutation). We illustrate these two operators using a partitioning problem with a contiguity constraint. The example problem partitions 4 contiguous *zones*) on a 4x4 grid, where each cell is a problem variable.

Figure 4.3 illustrates the steps in our path relinking-based crossover (PRCRX) algorithm. Subfigure (a) shows two parent solutions. The solution on the left is the *source solution*, S while the solution on the right is the *target solution*, T. The colors represent different zones. We first "overlap" these two solutions (shown in subfigure (b)), and give a *zone label*, $s \bullet t$, to each cell, where s is the cell's zone assignment (here, representing the cell color) in S and t is the zone in T. This creates a set of connected components. Cells in each component are connected and share the same zone label. Call each connected component a *group*, G. Adjacent groups have different zone labels, but non-adjacent groups may have the same zone label, depending on the shape of s and t.

Suppose this is a k-partition problem. The next step is then to pick k seed groups, where each group has a unique t in its zone label. For example, we could simply pick those groups with zone label $z \bullet z$, for $z = 1, \ldots, k$. Once the seed groups are identified, the solution distance is defined to be the difference between the total number of cells and the number of cells in the seed groups. In Figure 4.3(c), the distance, d, is $16 - 9 = 7$, which means that, at most, 7 moves are needed to "walk" from the source solution to the target solution. The 7 mutable units form k *candidate sets*, each indexed by the *label* t. A candidate set, C_z, is the complement of the corresponding seed group G_z, i.e., $G_z \cup C_z = Z_z$, where Z_z is zone z. That is to say, C_z comprises all of the remaining units in zone z in T. In Figure 4.3(c), we can see that the 4 candidate sets are 1 yellow, 3 red, 1 blue, and 2 green cells in the target solution, marked with the letters (Y–yellow, R–red, B–blue, G–green). The solution distance can be defined using G and Z as follows.

> **Definition**. The solution distance, d, between the source solution S and the target solution T is defined as $d = \sum_{z=1}^{k} |C_z| = \sum_{z=1}^{k} |Z_z - G_z|$.

One way to begin the relinking process or the walk between the two solutions is to first choose a particular source zone, i.e. a set of units whose zone label share the same s. In each *move*, or step of the path, we convert a cell from zone s to zone t, reducing the solution distance, d, by 1. Subfigures 4.3(d–f) illustrate the intermediate moves in a path relinking process that translates the source solution (subfigure (c)) to the target solution (subfigure (g)). To move from subfigure (c) to subfigure (d), we convert all of the cells labeled Y to the yellow zone. To move to the solution shown in subfigure (e), we convert the cells labeled R to the red zone. We convert the cells labeled B to the blue zone to obtain subfigure (f). Finally, we convert the cells labeled G to the green zone, which completes the path. Here, our path of length 4 travels the distance of 7 and generates 4 intermediate solutions (as colored).

There are many ways in which the path can be constructed. One could, for instance, move each candidate cell individually, generating 7 intermediate solutions (6 new). Alternatively, we can build a path with different step sizes at each move, as long as each move does not violate spatial constraints. The *path length*, which is defined as the number of spatial moves, is flexible, but also constrained by the number of mutable candidate cells available. It is possible that a particular order of candidate set/unit visits is infeasible, which is acceptable because the purpose of the walk is to generate intermediate solutions, not necessarily to end at T.

In the construction of any path, however, an important consideration is to properly deliberate seed groups before the relinking process begins to avoid the unnecessary computation

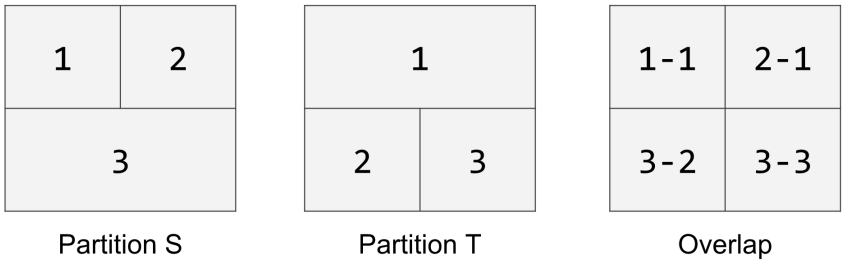


Figure 4.4: Zone labeling issue. Two seed groupings with unique t labels are possible: $(1 \bullet 1, 3 \bullet 2, 3 \bullet 3)$ or $(2 \bullet 1, 3 \bullet 2, 3 \bullet 3)$. However, neither has unique S labels. The expansion to T zones with any of the two seed groupings has two undesirable effects for finding new solutions with better fitness along the way. First, zone 1 and 2 in partition S will eventually merge unless the expansion is terminated early. Second, zone 3 in S is either stuck, as the case in this figure, or inappropriately expanded both zone 2 and 3 in T.

that arises from generating infeasible intermediate solutions. For example, recall that when we created the k seed groups, we *chose* those groups with identical s and t zone labels. This is but one of a set of possible choices. We could have chosen any $s \bullet t$ label where the k seed groups have k unique s and k unique t labels. A complication may arise here since there does not always exist such a unique zone labeling in the overlap, as we show with the example in Figure 4.4. The consequence when unique zone labeling for both S and T does not exist, is that two source zones will merge as we expand the groups, G_t, because there are at least two seed groups that share a source zone. In the expansion of the two seed groups to target zones, it is highly likely that the fitness of the intermediate solutions is worse. Accordingly, in our implementation, we seek to identify unique zone labelings and to minimize groups with the same source zone labels. A path may also be incomplete because of the contiguity constraint. If a group, G_t, or units in C_t are poorly selected, it may be impossible to expand G_t. Figure 4.5 illustrates such a scenario.

Unfortunately, it is difficult to know beforehand whether a seed grouping or a particular selection would lead to these problematic scenarios. This phenomenon affects the search space size, but the effect on fitness improvement is unknown. Furthermore, sophisticated algorithms for preventing such scenarios may be too costly to compute. Our current implementation detects if a potential move violates the contiguity of any source zone. If so, we choose a different move. If none of the candidate moves is possible, we simply terminate the relinking process.

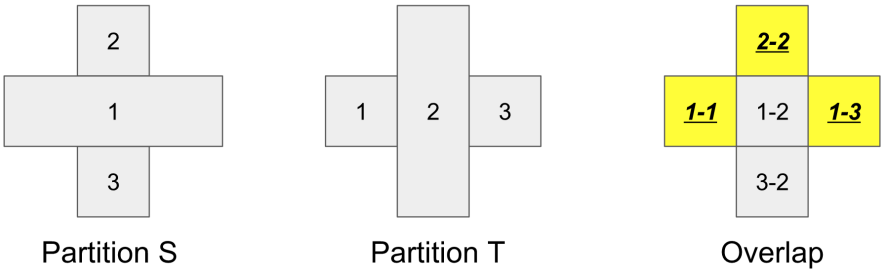


Figure 4.5: Seed group selection constrained by source zone contiguity. The selection of the yellow seed groups makes it impossible to main the contiguity of source zone 1. The expansion of group $G_2(2 \bullet 2)$ to group $G_2(1 \bullet 2)$ disconnects source zone 1. An appropriate zone labeling would select $G_2(1 \bullet 2)$ as the seed group.

4.2.5 Spatial Mutation through Ejection Chain

ECMUT (ejection chain-based mutation) builds on our previous spatial mutation operator (Liu, Cho and Wang, 2016). by generalizing the chained mutation steps into the ejection chain heuristic framework (Yagiura, Ibaraki and Glover, 2006). An ejection chain is a series of moves, each of which "ejects" an assignment of a problem variable to the next variable. This type of ejection forms a chain that can be cyclic or acyclic. Ejection chain is a generalization of local neighborhood functions such as shift (chain length=1) and swap (chain length=2). Non-spatial or spatial considerations may be utilized to determine the chain length and the units to eject in each move. For example, the order of moves in a chain can be defined by the adjacent vertices on a graph so that intermediate solutions are generated with each ejection move. Transforming a general ejection chain to one that is spatially explicit is straightforward when the neighborhood function is spatial.

Ejection chain-based mutation has an advantage over spatial designs such as location-based morph where one moves a few units to a new location (Xiao, 2008). The drawback of the latter is that, while the new area is able to contain the shape of the units, the change in the variable interdependencies is not preserved. Ejection chain-based mutation, on the other hand, can be designed to preserve both the spatial characteristics as well as the spatial dependencies.

Figure 4.6 provides an illustration of the ECMUT operator process. An ejection chain is cyclic. In our example, we use the cycle (yellow $\rightarrow$ blue $\rightarrow$ red $\rightarrow$ green $\rightarrow$), which goes from green back to yellow. Subfigure (a) shows a parent solution with 4 districts or zones, again denoted by color. At each step of the chain, a random set of contiguous cells from one zone is moved or "ejected" to its neighboring zone. Since our zone order is yellow $\rightarrow$ blue $\rightarrow$ red

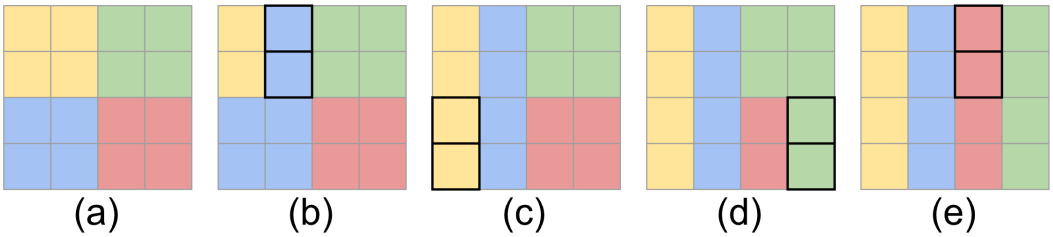


Figure 4.6: Ejection Chain Mutation (ECMUT) illustration.

→ green →, we first randomly move selected yellow border cells (subfigure (b)), then blue border cells (subfigure (c)), then red border cells (subfigure (d)), and finally green border cells (subfigure (e)), to complete a cycle. Each of the four intermediate solutions shown in subfigures (b)–(e) are valid solutions that are considered at the EA replacement phase.

4.3 Algorithms

We now detail how path relinking and ejection chain are integrated into the spatially explicit evolutionary computation (SEEC) methodology. The algorithms presented in this section apply to partitioning problems with spatial constraints. A partitioning problem divides n spatial units into k disjoint contiguous zones. In graph terminology, the goal is to generate k disjoint connected components on the adjacency graph that cover all n vertices. Rook adjacency leads to a clearly planar graph while queen adjacency may not lead to a planar graph. We employ a chromosome encoding for solutions and groups with operations that rely on the adjacency graph and its associated graph operations.

4.3.1 Spatial Crossover (PRCRX) Algorithm

Our spatial crossover method has the following characteristics:

- The overlap, assisted with unique zone labeling, creates a set of connected components that mix the spatial configuration of two parent solutions. Overlapped groups serve as the basic unit for crossover recombination.
- The seeding of k groups defines the solution distance, which is the maximum path length. Which k groups should be selected as seeds is a question whose answer is tightly related to the maintenance of spatial constraints and the search paths that lead to improved fitness.
- Group expansion grows seed groups to the target solution. The expansion is based on the selection of adjacent units in candidate sets.

- Each move in the expansion, i.e., the selection of a unit in a candidate set, is applied to the involved source zones and morphs the source solution into a series of intermediate solutions. The sequence of these moves forms the path. When the length of the path is equal to the solution distance, the relinking process transforms the source solution into the target solution. Which mutable units in the candidate sets should be chosen for expansion is determined by the fitness improvement of these intermediate solutions. Because repair operations are computationally costly, this expansion preserves contiguity.
- The possibility of generating different and new solutions lies in the flexibility of choosing different candidate sets and different units in them at different steps of the path.
- Randomization is invoked at the group seeding, ordering of the candidate set, and the selection of the adjacent candidate unit.

Algorithm 4.1: A general overlap operation.

```
function overlap(U, S, T) :
  foreach u ∈ U :
    i = S[u]; j = T[u] // zone assignment in the two solutions
    X_{i,j} = X_{i,j} ∪ {u}
  return {X_{i,j} | X_{i,j} ≠ ∅}
```

Algorithm 4.2: Spatial overlap operation.

```
function spatial_overlap(U, S, T) :
  X = [0] // X[j] = 0, j = 1..n
  i = 1 // group index
  Q = ∅
  foreach unit u in U:
    if u has been assigned a group:
      continue
    X[u] = {i} // create a new group
    Q.enqueue(u)
    while Q ≠ ∅ :
      u = dequeue(Q) // fetch a unit from Q
      N_u = {v | v is a neighbor of u on the adjacency graph}
      foreach v ∈ N_u:
        if v has been assigned a group:
          continue
        if S[u] = S[v] and T[u] = T[v]: // same zone labels and contiguous
          X[v] = X[u] // same group
          Q.enqueue(v) // recursive search to v's neighbors
    i = i + 1;
  l = i;
  build G = {G[i], i = 1..l} from X
  return X and G
```

The path relinking process has three primary steps: overlapping, group seeding, and path building. A general overlapping algorithm, outlined in Algorithm 4.1, applies to all combinatorial optimization problems with chromosome encoding. It returns the overlap, X, as a collection of group sets with a time complexity of $O(n)$. However, since a group in X is indexed using a zone label, it does not differentiate two groups with the same zone label, which is possible when two zones from S and T intersect and result in multiple connected components. Algorithm 4.2 modifies the basic overlap operator so that it is now spatially explicit. This algorithm is based on the well-known connected component labeling algorithm (Cormen et al., 2009), but has been adapted to our graph representation of the problem. The algorithm applies breadth-first search (BFS) and returns the overlap, G, and its chromosome encoding, X, where $X[j]$ is the group assignment of unit j. The time complexity is $O(m+n)$, where m is the number of edges on the adjacency graph. For planar graphs, m is a constant factor of n. Other efficient implementations may use the Union-Find algorithm (Cormen et al., 2009) within $O(n\alpha(n))$, where α is a very slow-growing inverse of the rapidly increasing Ackermann function.

The group seeding step appropriately selects k seed groups from $\{G[i], i = 1, \ldots, l\}$, where l is the number of groups derived from X. The solution distance between partition S and T is $l-k$. The *overlap* algorithm creates at least k and at most $\min(n, k^2)$ groups. If the k seed groups are small in size, i.e. do not comprise many units, the solution distance, d, in number of units, is large. In this case, the path building process is computationally costly. If the k selected seed groups are large in size, we have a different, but also problematic scenario where the number of available moves is limited and consequently less likely to generate desirable new solutions along the path. This problem, however, is ameliorated in large scale problems. In our empirical work, we found that selecting k large seed groups with unique zone labeling is as effective as random group selection but more efficient because of the shorter solution distance, d. The seeding step returns seed groups, G_t, and their corresponding candidate sets, C_t, for $t = 1, \ldots, k$.

Once seed groups have been determined, we begin the path building process. From the target solution point of view, path building is equivalent to seed group expansion toward T. From the source solution point of view, path building can be seen as the construction phase for intermediate solutions. Algorithm 4.3 is a general path relinking algorithm based on an overlap of two solutions. It applies to non-spatial problems through the expansion of the G_t groups and the associated update to the source solution S. New solutions are generated by flipping a randomly selected unit in C_t to the assigned value of G_t variables. Without

modification, Algorithm 4.3 can solve set partitioning problems. With modifications to how C_t is constructed, the algorithm can be applied to assignment problems without the partition requirement. In this way, the coupling of overlapping and path relinking (or ejection chain) provides a new EA crossover choice.

Algorithm 4.3: PRCRX0: a general path relinking crossover algorithm.

```
1  function flip(S', G_t, C_t) :
2    randomly select a unit u ∈ C_t
3    G_t = G_t ∪ {u}
4    S'[u] = the source zone of G_t
5    C_t = C_t − {u}
6    return S';
7
8  function prcrx0(S, T) :
9    X = overlap(S, T)
10   {G_t},{C_t} = seedKuniq(X, k) // build seed groups and candidate sets
11   d = n − Σ_{t=1}^{k}(|G_t|) // solution distance
12   S_best = S' = S;
13   for i = 1..d − 1:
14     randomly select a zone z in T
15     flip(S', G_t, C_t);
16     evaluate S';
17     if S' is better than S_best:
18       S_best = S';
19   return S_best
```

Notice that our algorithm expands the G_t groups unit by unit, not group by group. Since the unit is the finest level of granularity for path relinking to operate on, adapting to expand an arbitrary number of mutable units in C_t for a single move is straightforward. Moreover, expanding a set of units is more efficient because this reduces the number of solution evaluation steps at line 16. The tradeoff is that unit level expansion permits greater exploration of the search space.

Algorithm 4.4 adapts Algorithm 4.3 from a general path relinking algorithm to a spatial path relinking algorithm. A concern with Algorithm 4.3 is that the $flip()$ function may produce disconnected components. Algorithm 4.4 addresses this problem by building paths based on spatial relationships represented on the adjacency graph and graph search methods. It conducts seed group expansion on neighboring candidate units and ensures contiguity by maintaining a second adjacency graph for the zone boundary units.

Algorithm 4.4: PRCRX1: a spatial path relinking crossover algorithm.

```
1  function prcrx1(S, T)
2    N; // adjacency graph for all the n units. N(u) returns u's neighbors
3    X = overlap(S, T) // build overlap
4    {G_t},{C_t} = seedKuniq(X, k) // build seed groups and candidate sets
5    d = n − Σ_{t=1}^{k}(|G_t|) // solution distance
6    S_best = S' = S;
7    foreach zone z in S': // establish boundary unit adjacency graph for contiguity check
8      build adjacency graph BG_z for boundary units in z
9    foreach zone z in T: // build initial adjacent unit set to G_z as candidate units
10     AC_z = ∅
11     foreach u in G_z: // a candidate unit neighbors G_z but in C_z
12       AC_z = AC_z ∪ {v|v ∈ N(u) ∧ v ∈ C_z}
13     zsgt_z = the zone of the initial G_z in S // the zone a mutable unit will move to
14   path = []; pathlen = 0;
15   do
16     zseq = a random sequence of size k to visit each zone in T
17     foreach z in zseq:
18       mu = 0 // the mutable unit
19       cu_best = 0 // stores the best candidate unit in C_z
20       foreach cu ∈ AC_z:
21         zs = S[cu] // the zone of cu in S, not in S'
22         if zs = zsgt_z: // same zone, no effect; but update G_z, C_z, AC_z
23           mu = cu; break;
24         else // peek to see if it is mutable and with fitness improvement
25           check BG: whether adding cu to zsgt_z disconnects a zone in S'
26           if contiguity is maintained:
27             evaluate fitness of S' if cu is moved
28             if the cu move leads to a better solution:
29               cu_best = cu
30       if mu = 0: // not ineffective move
31         if cu_best > 0 // found an effective mutable unit
32           mu = cu_best
33       AC_z = AC_z − {mu}; expand AC_z
34       G_z = G_z ∪ {mu} // update G_z
35       C_z = C_z − {mu} // update C_z
36       if mu is effective:
37         update BG // update boundary unit adjacency graph of S'
38         S'[mu] = zsgt_z // update the intermediate solution
39         path[pathlen++] = mu // record the move on the path
40         if S' is better than S_best:
41           S_best = S';
42   while there was successful expansion on any G_t
43   return path and S_best
```

As illustrated in Figure 4.5, at the time of seed group expansion, we must ensure that the addition of a mutable unit does not disconnect the source zone. One way to check zone contiguity is to count the number of connected units in the zone, starting from a randomly picked unit. If the count before the mutable unit removal is not equal to one more than

the count after the removal, the zone is broken. This is an easy though computationally expensive check that requires $O(d \times \frac{n}{k} \times \mu)$ time for k zones, where μ is the average degree of connectivity of each unit. Although μ is not large on a planar graph, this function must be called for each mutable unit. A more efficient way to ensure contiguity is to check whether the boundary units (using queen neighborhood) are connected only before and after unit removal. This check takes $O(d \times \phi(n,k) \times \mu')$, where function ϕ, which depends on the shape of the zones, estimates the number of boundary units in a zone and μ' is the average degree of connectivity among boundary units, which is much smaller than μ. Compact shapes have small ϕ values.

Algorithm 4.5: path_optimize: greedy search on an existing path for better solutions.

```
1  function path_optimize(S', T, G_t, path1, path1len)
2    S_best = S'' = S;
3    foreach zone z in S'': // establish boundary unit adjacency graph for contiguity check
4      build adjacency graph BG_z for boundary units in z
5    foreach z in T:
6      muindex_z = index of the first moved unit in G_z
7    path2 = []; path2len = 0;
8    while path2len < path1len:
9      mu = 0; zsmu = 0
10     foreach zone z in T:
11       if muindex_z = len(G_z) // group is exhausted
12         continue
13       cu = G_z[muindex_z] // get a moved unit and re-evaluate
14       zsgt = S''[cu]
15       check BG: whether adding cu to zsgt_z disconnects a zone in S''
16       if contiguity is maintained:
17         evaluate fitness of S'' if cu is moved
18         if the cu move leads to a better solution:
19           mu = cu; zsmu = zsgt
20     if mu = 0 // no zones have feasible moves, end of the loop
21       break
22     update BG // update boundary unit adjacency graph of S''
23     S''[mu] = zsmu // update the intermediate solution
24     path2[path2len] = mu // record the move on the path
25     if S'' is better than S_best:
26       S_best = S''
27     path2len ++
28     muindex_z ++
29   return path2 and S_best
```

PRCRX1 randomizes the order in which the T zones are visited as well as the order for adding adjacent units in candidate sets. This randomization diversifies the decision

space search. The resulting path, called *path 1*, can be further optimized with little cost. Algorithm 4.5 implements an optimization strategy by scanning path 1 in a greedy fashion. We begin with k seed groups, formed at the beginning of the relinking process in PRCRX1, and check the k G_t groups to identify the best mutable unit to add to a new path. This process requires $O(pathlen)$ time and creates another path, called *path 2*, along which another best solution is found. This process allows us to identify the best choice from the two paths as the output of path relinking.

4.3.2 Spatial Mutation (ECMUT) Algorithm

We also design our mutation algorithm within the same path relinking framework as our mutation operator. The primary difference between the ejection chain-based mutation and the path relinking-based crossover is that, in the mutation operator, the initial solution and the target solution are the same solution. That is, the operator searches a neighborhood along a path that begins and ends at the same solution. The chaining process is similar. In ECMUT, we alternate the zones to visit, and each visit identifies a number of mutable units among which at least one is on the zone boundary. These units are then moved to a neighboring zone, identified from the zone adjacency graph. Algorithm 4.6 shows how a general ejection chain mutation algorithm can be adapted from path relinking while Algorithm 4.7 further modifies this general algorithm to incorporate spatial considerations. The implementation of Algorithm 4.7 is detailed in Liu, Cho and Wang (2016). Depending on the particular spatial configuration characteristics, ECMUT may or may not be able to produce cyclic chains, which does not harm the optimization process.

Algorithm 4.6: ECMUT0: A general ejection chain-based mutation algorithm.

```
1  function ecmut0(S, ecLength, cyclic) :
2    seq = a random sequence of 1..n // n is the number of units
3    z_inject = S[seq[1]] // zone assignment of the first ejecting unit
4    for i: 2 − max(n, ecLength) :
5      z_eject = S[seq[i]] // eject the current unit
6      S[seq[i]] = z_in // unit re−assigned to the zone from the previous ejecting unit
7      z_inject = z_eject // inject to next
8    if cyclic is True :
9      S[seq[1]] = z_inject // cyclic ejection chain
```

Algorithm 4.7: ECMUT: a spatial ejection chain-based mutation algorithm.

```
function ecmut_contig(U, S, ecLength, blocksize) :
  // determine the order of zone visit. ecLength is the length of the ejection chain
  seq = a random sequence of 1..ecLength
  Solutions = ∅ // Solution set that holds improved solutions during the ejection chain
  fitness0 = fitness(S)
  // Iterative ejection chain
  q = 1 // MU sequence index
  for i: 1..ecLength // Initialize units in each zone
    z_eject = (seq[i]%k) + 1 // Zone index of the eject zone; k is the number of zones
    Z_eject = Z[z_eject] // Unit set of the eject zone
    z_inject = select_receivingZone(z_eject) // Select a zone to inject the ejected units
    Z_inject = Z[z_inject] // Unit set of the inject zone
    Ub_inject = {u | u is a boundary unit of zone z_inject}
    // Select a set of contiguous units, Δ_q,
    // where Δ_q ⊆ Z_eject, |Δ_q| ≤ blocksize and Δ_q is adjacent to Z_inject
    Δ_q = select_mutable(Z_eject, Ub_inject, blocksize, Z_eject)
    if Δ_q = ∅ :
      continue
    foreach u ∈ Δ_q : // Move these units to the inject zone
      S[u] = z_inject
    if fitness(S) is better than fitness0 :
      Solutions = Solutions ∪ {S}

  return Solutions

function select_mutable(P, B, maxCount, C) :
  // P: a pool of contiguous units
  // B: neighboring units to P, each sharing a common border to at least one unit in P
  // maxCount: max number of movable units to select, as the stopping rule in the search
  // C: An intermediate solution in which each zone's contiguity holds after MU selection
  U_b = { u | u ∈ P and u is adjacent to at least one unit in B} // Border units in P to B
  u_0 = a unit randomly selected in U_b
  M = {u_0} // Initial movable unit set
  // Randomized recursive traversal of the adjacency graph of P to select movable units
  randUnitSearchRecur(P, M, maxCount, C)

  return M

function randUnitSearchRecur(P, M, maxCount, C) :
  // M: reference to the modifiable set of movable units
  if maxCount = 0 : return // Recursion exit
  // Find neighboring units not yet included in the movable unit set
  N = { neighboring units of M in P }
  if N = ∅ : return // Recursion exit
  m = a random number in [1..min(|N|, maxCount)]
  M' = { up to m randomly selected units from N, that keep C contiguous}
  M = M ∪ M'
  randUnitSearchRecur(P, M, maxCount − m)
```

4.4 Evaluation

We evaluate PRCRX and ECMUT with political redistricting, a real-world spatial partitioning application. The redistricting problem can be formulated as an SO problem that translates the legal requirements of redistricting (e.g., contiguity, equipopulation, and competitiveness) into spatial and non-spatial constraints and objectives. In this evaluation, partition and contiguity are integrated into the problem solving. We evaluate our spatial EA performance in four ways. First, we compare the solution quality and performance with existing heuristics. Second, we compare the performance gain when PRCRX is used in addition to ECMUT. Third, we study the performance characteristics of PRCRX. Finally, we conduct tests in a parallel computing environment to examine the algorithm's scalability and convergence properties.

4.4.1 Implementation and Case Study

Liu, Cho and Wang (2016) developed PEAR, a high-performance computing tool for redistricting. We adopt the same problem formulation, EA/PEA configuration, and EA population initialization, selection, and replacement operators. ECMUT was implemented and evaluated in Liu, Cho and Wang (2016). In that implementation, two chains were applied in order. First, the general ECMUT was applied to search the decision space. Second, an ejection chain was applied to the output solution from the previous ECMUT for feasibility improvement. Chained moves are designed to tune the values of constraint measures under feasibility thresholds. In this chapter, we implement a new version of PEAR, PEAR++, to include PRCRX, which is the focus of our evaluation. In this new implementation, a random restart feature is added to avoid early convergence where the local population becomes sufficiently homogeneous that new solutions are hard to identify.

PEAR++ is implemented in C++, and compiled using GCC 4.9.2 and the MPICH 3.1.4 library. To enable asynchronous migration, it uses MPI non-blocking functions (i.e., *MPI_Ibsend()*, *MPI_Iproble()*), and regular *MPI_Recv()*. The SPRNG 2.0 C library provides a unique random number sequence for each MPI process. Performance tests are conducted on the ROGER supercomputer. Each node on ROGER is configured with the Intel Xeon E5-2660 processor (2.6GHz, 20 cores/node), 256GB memory. The cluster is connected by a high-speed network with 40Gb/s switches in the core and 10Gb/s uplinks to each node.

Our particular application examines the redistricting problem in North Carolina at the voter tabulation district (VTD) level. The GIS census data includes information on the polygon shape, population, election, and party registration information for each district in

Table 4.1: PEA parameter setting

Fitness	$0.2 \times equipopulation + 0.8 \times competitiveness$
PRCRX output	The better solution from $prcrx1()$ and $path\_optimize()$
Population size	200
Selection	Binary selection
Initial population	80% by region border; 20% by administration border
Elitism	On
Homogeneity check interval	20,000 iterations
Homogeneity threshold	95% population's solution distance $< 10\% \times n$
Export/import interval (PEA)	100/50
Migration rate (PEA)	2
Sending parallelism (PEA)	4

2011. There are 2,690 VTD units after preprocessing. The rook and queen neighborhood matrix are obtained using open source GIS libraries, PySAL (http://pysal.org) and GDAL (http://gdal.org). For each of the experiments reported, PEAR++ is configured with the parameters specified in Table 4.1. Detailed description of redistricting and PEA parameters can be found in Liu, Cho and Wang (2016).

4.4.2 Comparison with Other Heuristics

We first compare the sequential version of PRCRX and ECMUT to heuristics designed by others for the redistricting problem. These heuristics include simulated annealing, greedy algorithm, tabu search, GRASP, and GRASP with contiguity support. We have previously compared ECMUT to these other algorithms (Liu, Cho and Wang, 2016). In Table 4.2, we show those results augmented with the results from our spatial path relinking operator. As we can see, of the other algorithms, the GRASP (contiguous) algorithm produced the best result. It ran for a total of 20 hours, identifying its best fitness, 0.0411, near the 5 hour mark. The "ECMUT snapshot" line shows that ECMUT handily outperformed these other algorithms, identifying a solution with a fitness value of 0.0145 in 1,555 seconds. In addition, unlike the GRASP (contiguous) algorithm which then ran for another 15 hours without finding a better solution, ECMUT continued to improve over the course of an hour. Its best solution with fitness 0.0145, is shown on the line labeled "ECMUT best." These are impressive results, but our EA improved even more significantly when the PRCRX operator was included. The performance with both operators is shown in the lines labeled "PRCRX+ECMUT." The snapshot shows that PRCRX+ECMUT surpassed the best solution identified by GRASP

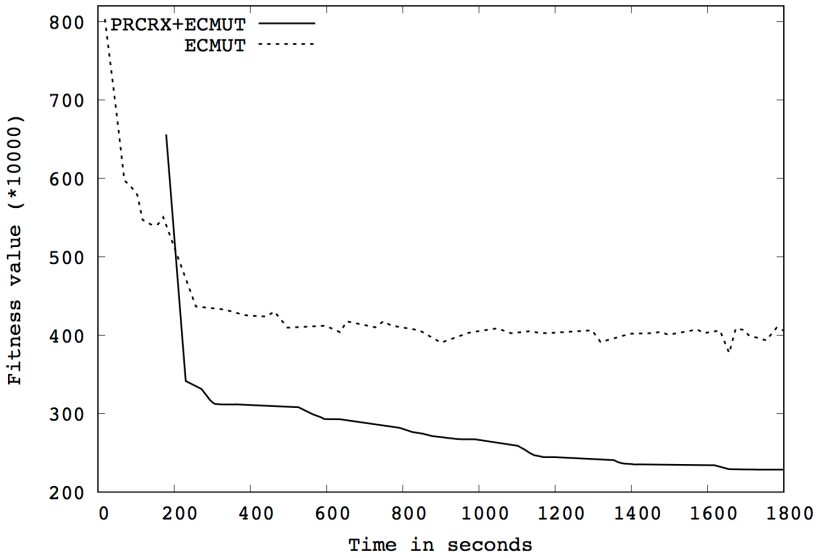


Figure 4.7: Impact of PRCRX on fitness change. Lower fitness value indicates better solution quality.

(continuous) in just 148.96 seconds. The algorithm continued to run for the rest of an hour, at which time it identified a solution with a fitness value of 0.0098, shown on the line labeled "PRCRX+ECMUT best," with notably impressive result on both of the fitness components, equipopulation (0.0015) and competitiveness (0.0098).

The iterations per second for PRCRX+ECMUT is somewhat elevated due to some early convergence issues. PRCRX and the feasibility improvement ECMUT are the most costly operators. When early convergence occurs, solutions in the population become too homogeneous, which affects the PRCRX crossover (solution distance is too short). Also, most of the solutions in the population are feasible, so there was no need to perform feasibility improvement. As a result, iterations accumulated without PRCRX and the feasibility improvement ECMUT. The iteration speed measured at the time of surpassing GRASP (contiguous) is more accurate at 5.19 iterations per second, indicating that PRCRX is five times slower than ECMUT, compared to ECMUT's 30.56 iterations per second. However, given length of the relinking path, numerical performance of PRCRX remains satisfactory.

We further examine the impact of PRCRX on the evolutionary process by sampling new solutions generated in both the ECMUT and the ECMUT+PRCRX runs. The change in fitness for the feasible elite solutions is shown in Figure 4.7. At the start of the EA, PRCRX is somewhat disruptive. PRCRX sufficiently diversified the search so that the first feasible solution occurred later than ECMUT. Shortly thereafter, however, with more

Table 4.2: Performance comparison.

	Solution quality			Cost			Cost per improvement		
	Best fitness	Competi-tiveness	Equal population	Time (in seconds)	Iterations	Iterations per second	Improve-ments	Iterations	Time
Simulated Annealing	0.1237	0.0696	0.3402	1,489.12	1,472	0.99	130	11.32	11.45
Greedy	0.0980	0.0659	0.2264	3,817.91	186,619	48.88	858	217.50	4.45
Tabu	0.0984	0.0658	0.2288	1,968.94	92,659	47.06	794	116.70	2.48
GRASP (default)	0.0980	0.0659	0.2264	3,910.87	186,619	47.72	858	217.50	4.56
GRASP (contiguous)	0.0411	0.0491	0.0093	19,140.91	320,386	16.74	2075	154.40	9.22
ECMUT snapshot	0.0403	0.0425	0.0316	1,555.64	45,358	29.16	44	1030.86	35.36
ECMUT best	0.0145	0.0117	0.0256	10,784.55	329,572	30.56	142	2320.93	75.95
PRCRX+ECMUT snapshot	0.0383	0.0410	0.0272	148.96	774	5.19	18	43.00	8.27
PRCRX+ECMUT best	0.0098	0.0119	0.0015	3,558.72	351,949	98.89	475	740.94	7.49

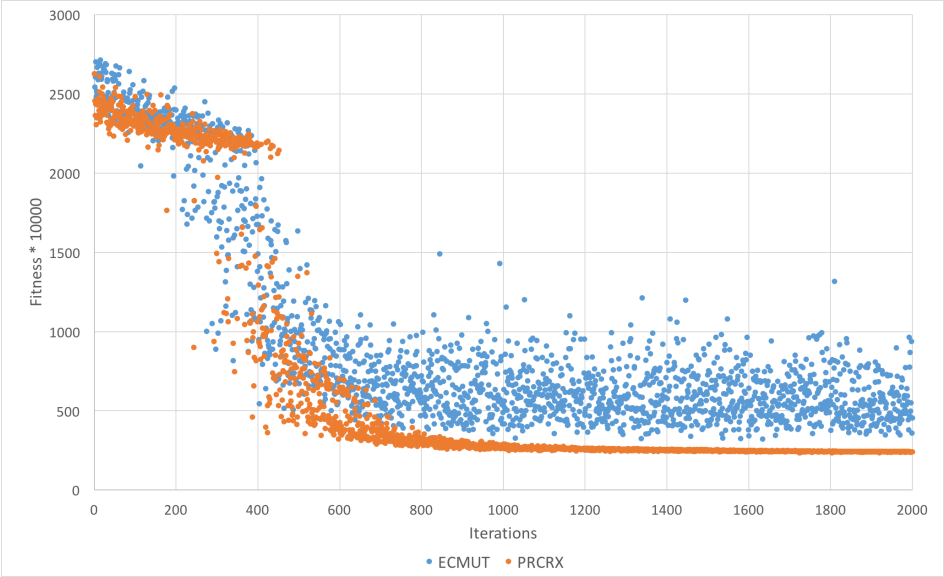


Figure 4.8: Performance comparison: PRCRX vs. ECMUT in the first 2000 iterations. For minimization problems, lower fitness values mean better solution quality.

feasible solutions in the population, the path relinking process becomes very effective. Several apparent fitness improvements result from random restarts, which keep the top 10% solutions in the current population and inject 90% random solutions. PRCRX appears to be more adaptive to random starts than ECMUT.

4.4.3 PRCRX vs. ECMUT

To examine the difference in solution quality between PRCRX and ECMUT further, we reconfigured the EA so that PRCRX and ECMUT take the same parent solutions from the population. Figure 4.8 shows the solution fitness (feasible and infeasible) from the first two thousand iterations where PRCRX outperformed ECMUT 91.65% of the time. In a broader sampling of 20,000 iterations, PRCRX produced a better solution quality 99.16% of the time. While ECMUT was slow to improve fitness, PRCRX was quickly trapped in early convergence (shown in the right part of the orange curve) and could not make much progress until the first random restart. In addition, while ECMUT may outperform PRCRX at the beginning of the search, this pattern is unlikely to continue as fitness improves. Interestingly, around the 400th iteration, there is an obvious and significant fitness improvement phase, which indicates a pivot point that is often described as the phase transition point in an

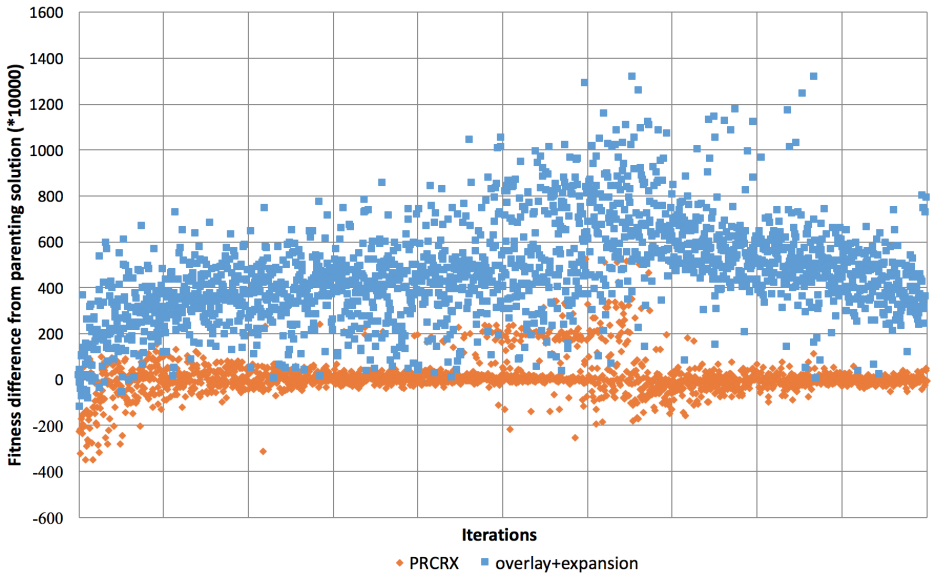
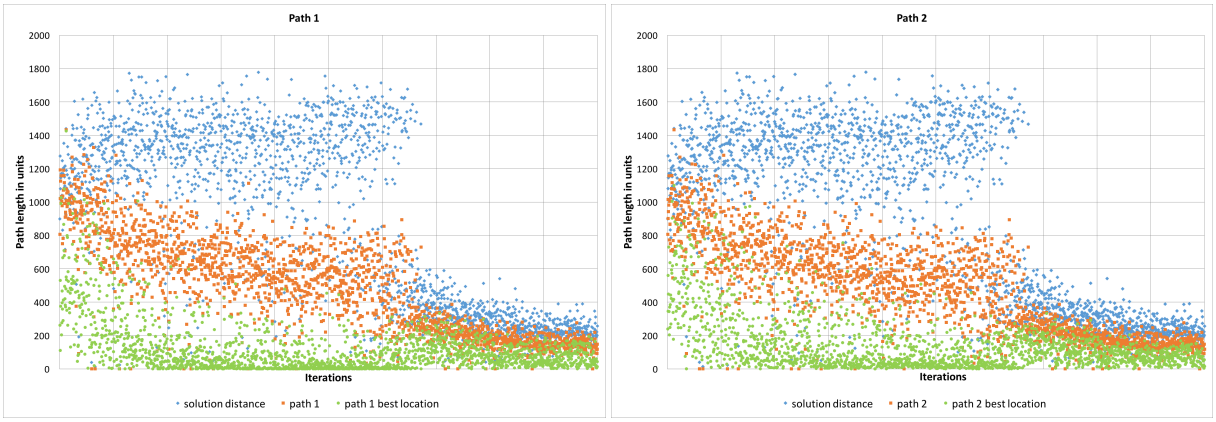


Figure 4.9: Performance comparison: PRCRX and the basic overlay+expansion. Both take the same input. Fitness difference between the input and output solution is plotted. Negative difference value means fitness improvement.

optimization process. Both PRCRX and ECMUT captured it because they receive the same parent solutions.

4.4.4 PRCRX vs. Basic Overlap-based Recombination

Overlapping is an effective way to combine two spatial solutions from which one generates a set of connected components of finer granularity for recombination. Based on the resulting overlap, different methods can be developed to generate new solutions. In this experiment, we compare path relinking-based crossover with a simple overlap-based recombination method, which simply expands the k overlap groups into k zones. The results are shown in Figure 4.9. The fitness change from the same parent solutions is plotted. It is obvious that simply recombining overlap groups is not an ideal strategy. PRCRX, on the other hand, shows effective performance in improving fitness of the population. One possible reason might be that PRCRX leverages the overlap groups for recombination, but the main source of fitness improvement comes from path relinking, in which the path links two parents and the walk on the path generates child solutions that are close to two parents, preserving alleles of both. This speculation reminds us of the difference between crossover and recombination, namely, the inheritance of alleles from parent solutions. Recombination methods may not consider the inheritance part.



(a) Path 1 (b) Path 2

Figure 4.10: Path length analysis. Left: the solution distance (blue), path 1 length (orange), and the location of the best solution on path 1 (green) are plotted. Right: the solution distance (blue), path 2 length (orange), and the location of the best solution on path 2 (green) are plotted.

4.4.5 Path analysis

Figure 4.10 plots the solution distance, path length, and the location offset of the best solution found on the path for the first 2000 PRCRX calls in a run. Path 1, from *prcrx*(), and the optimized path 2, from *path_optimize*(), show similar patterns. Notice the sudden drop of solution distance and path length around iteration 1,200. This may indicate a pivot point at which the population becomes more homogeneous. It may also be the turning point from a diversified search to an intensified search. Second, the gap between solution distance and path length exists throughout the test runs, which indicates that spatial constraints might be a blocking factor that prevents the walk from the source solution S to the target solution T. Third, in PRCRX, the better of the two parent solutions is chosen as the target solution T. Intuitively, intermediate solutions should improve when walking from S to the better T and the best solution might be found closer to T. However, the location of the best solution (green points) along the path did not seem to follow a pattern, suggesting that the distribution of better solutions is more dispersed.

We also compare path 1 and 2 by seeing whether the best solution tends to emanates from path 1 or 2 in the iterations before the first random restart. On average, 68.83% of the best solutions originate from path 2, which makes sense because path 2 is built upon path 1. However, there are still 31.17% best solutions arising from path 1, indicating the exploration of path 1 in *path_optimize*() may be affected by spatial constraints which inhibit further walk on path 1.

Table 4.3: Weak scalability, which measures how long it takes (in seconds) to reach multiple fitness thresholds using different number of processors in parallel runs. A measure is left empty if a run could not reach to the threshold in one hour.

Number of processors	Fitness thresholds							
	1000	750	500	250	200	150	100	80
10	77.05	93.38	158.19	158.19	278.02			
20	13.26	78.78	119.90	204.28	251.64	563.06		
40	76.46	76.46	125.61	188.10	253.12	995.91	2043.65	
80	25.99	67.45	154.85	242.60	377.14	921.40	1769.73	
160	0.66	88.95	124.24	165.59	200.18	368.39	846.78	998.98

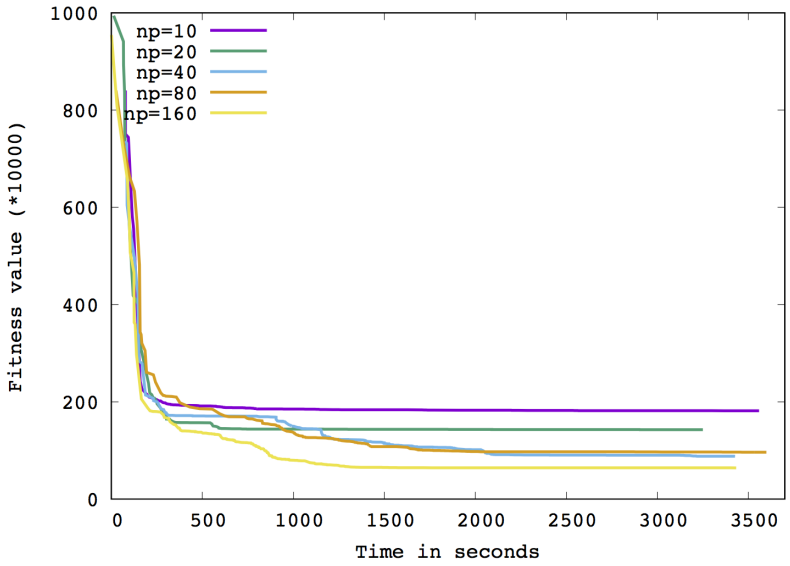


Figure 4.11: Parallel EA convergence.

4.4.6 PRCRX in Parallel Computing Environment

The aforementioned experiments come from the sequential PRCRX algorithm. We now examine its performance in a parallel computing environment. Here, we ran PEAR five times on ROGER, each using 10, 20, 40, 80, and 160 processor cores, respectively. Each computing node utilized 10 MPI processes for a total of one hour.

A weak scaling test measures a parallel code's capability of doing more numerical work when more computing power is added. As we increase the number of processors, the size of the global population increases as well. We measure the time taken to reach multiple fitness thresholds by using different number of processors. The results are shown in Figure 4.11. Overall, as we employ more processors, PEAR++ is able to reach tighter fitness thresholds.

For the same tight thresholds (150, 100, 80), adding more processors reduced the computing time. There are variations in this pattern, which is normal for a stochastic algorithm. For example, the run with 20 processors exhibited impressive performance.

Figure 4.11 illustrates detailed fitness change over time. Compared to sequential runs, parallel runs are more effective, but also more prone to early convergence. In fact, the addition of the random restart strategy in PEAR++ aimed to alleviate the early convergence issue. While we still observe early convergence using 10 and 20 processors, random restart seems to work well on large runs, showing steady fitness improvements.

4.5 Conclusion and Discussion

To avoid the disruption of linear combinations on non-linear decision space in spatial optimization, *an effective EA must leverage the underlying spatial configuration to guide the incremental iterations in searching for new solutions*. Meanwhile, *the principles of crossover and mutation have been shown to be effective and are able to be adapted for a spatial framework*. We propose SEEC, a novel spatially explicit evolutionary computation framework where the classic crossover and mutation operators are replaced with the intelligent path relinking and ejection chain heuristic operators which allow us to design guided spatial moves around the underlying non-linear decision space. Our two spatial recombination operators, PRCRX and ECMUT, satisfy spatial constraints and generate new feasible solutions by chaining spatial moves through randomization and optimization on solution fitness. This design allows the development of particular spatial moves and chaining strategies to incorporate domain-specific problem insights in spatial optimization.

The adoption of path relinking and ejection chain heuristics in the broader context of combinatorial optimization is not new. Yagiura, Ibaraki and Glover (2006) successfully combined path relinking and ejection chain to solve the non-spatial combinatorial optimization Generalized Assignment Problem (GAP). The connections between EA crossover and path relinking and the hybrid use of path relinking and EA methods have also been discussed in Glover (1994). Directly employing path relinking as the spatial crossover and ejection chain as the spatial mutation operator is, however, novel in resolving the major limitation of classic EA crossover and mutation in handling non-linear spatial structures in SO. To the best of our knowledge, this is the first approach of this kind which integrates path relinking and ejection chain in the EA recombination framework for spatial optimization.

Implementing PRCRX and ECMUT, which can be viewed as a spatial extension to path relinking and ejection chain heuristics, imposes serious algorithmic challenges. The relinking

and the chaining, respectively, are often linear combinations of simple shifting and swapping operations that can be expressed in a mathematical formulation in a straightforward way. When adapted to handle spatial constraints and objectives, we needed to develop sophisticated algorithms to devise spatial moves on spatial data structures such as an adjacency graph. Numerous geometric and topological operations are implemented as fast graph algorithms for the computational efficiency of the two operators. Examples include contiguity check, district shape intersection and union, and shape boundary relationships.

As a result, we successfully coupled path relinking and ejection chain principles, which share similar optimization roles with crossover and mutation, respectively, with a spatial representation of the decision space. The achieved evolutionary computation framework is evaluated by solving a partitioning problem with a contiguity constraint, which is considered the most challenging class of spatial optimization problems. The redistricting problem was solved by our framework and the computational performance and solution quality are compared with existing approaches in literature. Our solution showed more desirable effectiveness and efficiency. Indeed, the main source of performance gain is from the recombination operator level incorporation of spatial characteristics in the underlying problem. Our framework is seamlessly integrated into our parallel evolutionary algorithm library. The new path relinking operator provides an effective means for the recombination of solutions found on different processors, creating a mechanism to leverage local populations for global evolutionary computation and optimization.

Spatial constraints have a profound impact on the search properties of the relinking and chaining process. One interesting observation in our implementation is that due to the topological contiguity requirements in spatial variables and districts, a relinking process may not be able to entirely reduce the distance between the source and the target solution. A chaining process may not be able to produce a cyclic chain due to the availability of mutable units. In our current implementation, the definition of relinking path and ejection chain is relaxed to allow spatial recombination to proceed. Whether there is a complete relinking path and where there exists a cyclic chain are interesting and valuable research topics that we will pursue as next step.

Our approach creates a spatial recombination framework that allows further investigation and development of effective recombination strategies. The abstraction of spatial neighborhood function and the relinking and chaining provide basic building blocks for exploring different search orders at different granularities. Our framework can be developed to support more comprehensive spatial elements and fuse them into our heuristic search process.

PRCRX and ECMUT can be integrated into a general spatial EA framework for effective and efficient problem solving for large instances of spatial optimization. Algorithm 4.3 and 4.6 showed how path relinking and ejection chain can be employed as generic crossover and mutation operators. PRCRX and ECMUT modified these algorithms to be spatially explicit. This spatially explicit approach achieved convincing algorithmic and computational performance in our evaluations. Indeed, without specific data structures and routines to incorporate spatial knowledge into the optimizing process, it is difficult to navigate through the astronomically large decision spaces that characterize large SO problems.

Overlapping/overlaying is a basic technique to for combining two spatial solutions and generating a set of connected components with finer granularity for recombination. However, our observation is that it is difficult to relate the result of overlapping to guided recombination strategies. This may be because while overlapping mixes two solutions, the result is determined by shape and the location of shapes. It is unclear how the intersection of shapes at different locations preserve alleles of parent solutions. Path relinking leverages overlapping to obtain seeding groups, but bridges two solutions through incrementally constructed paths. These paths may preserve good solution traits from both parents. Furthermore, the recombination occurs along the path, allowing a controlled search for new solution generation. The experience with path relinking for SO tells us that overlapping-based recombination methods may not be classified as crossover operators if they are unable to link parent solutions at the new solution generation phase.

The PRCRX and ECMUT designs nicely generalize to an SO framework. The graph representation for spatial constraint handling in relinking path and ejection chain construction can be used to handle various spatial elements. The abstraction of chaining incremental spatial moves into path relinking and ejection chain is analogous to the nature of crossover and mutation in EA. How to design the local moves and build relinking paths and ejection chains can be solidified by extending our framework for a specific SO problem. The computational scalability of our PEA is evident.

The current implementation of PRCRX and ECMUT targets spatial partitioning problems where contiguity and partition are considered. We plan to incorporate more spatial elements in the framework, which will require enhancements to handle more complex spatial configurations that may require intensive topological and geometric processing. GPU computing may provide a good solution for conducting such computation without interfering with the EA process on CPU.

Finally, another research topic might extend the efficiency of PRCRX and ECMUT by avoiding early convergence issue. Here, one must consider that detecting population homogeneity is computationally expensive. Even the simple binary distance measure on p chromosomes requires the building of a $p \times p$ matrix and takes $O(p^2 \times n)$ to compute, where n is the number of variables. However, the solution distance and path length computed in PRCRX seem to be a promising indicator of population homogeneity. We currently draw parent solutions for PRCRX from the EA population. It might be a good idea to separate target solution set from the population and have explicit control on solution distance among the target solutions.

CHAPTER 5

CONCLUSION AND FUTURE DIRECTIONS

My research tackles the scaling problem of spatial optimization by investigating fundamental computational challenges and search strategy issues. The outcome is a scalable EA approach that combines massive parallel evolutionary computation and effective EA operators for spatial elements in optimization problems. A summary of my contributions is described in this chapter, followed by a discussion of vision for future.

5.1 Summary of Contributions

5.1.1 Scalable PEA Library

The work of the PGAP library Liu and Wang (2015) explores the computational bottlenecks of PEA scalability and proposes a scalable solution by eliminating the global synchronization barrier. Research questions **Q1-2** are answered by this work:

Answer to Q1. The global synchronization among PEA processes exposes the most significant computational bottleneck in PEA computation at the communication layer. The communication cost introduced by the synchronization can be as bad as 57.39% on 16,384 processors. Indeed, synchronization reduction is an important aspect for tackling exascale algorithm design which is listed as one of the top 10 challenges in exascale computing (Lucas et al., 2014).

Answer to Q2. An asynchronous migration strategy is proposed as a fine granularity of parallelism at PEA operator level to eliminate the synchronization bottleneck. This strategy improves message passing locality at the communication layer with the price of handling messaging buffers at application level. Two theoretical conditions are established to guide the appropriate configuration of PEA parameters to avoid buffer overflows. The introduction of asynchrony "obtain much higher factors of fine-grained parallelism as high-end systems support increasing numbers of compute threads" for large PEA runs (Ashby et al., 2010).

The establishment of our PEA library provides a scalable computing solution to exploit massive supercomputing power in the research presented in Chapters 4 and 3. In addition, the asynchrony brings an additional source of randomness from the unordered message passing and handling, which adds to the randomness that comes simply from the use of a parallel random number generator.

5.1.2 Large Spatial Optimization

In PEAR, a scalable evolutionary computation approach was developed to tackle large political redistricting problems. In developing the approach, our spatial PEA library was used to consider explicit spatial elements (e.g., contiguity, the hole-free property, adjacency, and compactness) and implicit spatial elements (e.g., the impact of population density on the grouping of voting districts) and provided a massively parallel computing approach for statistical analysis on output solutions. Research questions **Q3**–**4** are answered by this work (Liu, Cho and Wang, 2016; Cho and Liu, 2016):

Answer to Q3. We formulated political redistricting as an integer programming problem, in which legal requirements are translated into a series of explicit and implicit spatial elements and non-spatial elements. The generality of spatial constraint representation applied well in redistricting to capture constraints and objectives in contiguity, hole-free, protection of right groups, compactness, and equi-population. These elements are exploited in the crossover and mutation operators to effectively search the large variable space.

Answer to Q4. A set of redistricting measures for evaluating partisan gerrymandering is implemented in our approach to provide a quantitative method for detecting gerrymandering. A sampling mechanism is developed to create a sample pool at the size of billions through high-performance computing on Blue Waters.

5.1.3 Spatially Explicit Evolutionary Computation

A novel spatially explicit evolutionary computation framework is designed to overcome the drawback of randomization in classic EA by using strategically directed crossover and mutation operators. We evaluated adjacency, contiguity, compactness, and partitions in the proposed EA operators upon their effectiveness and efficiency in accelerating the EA process for large SO problems. In this framework, classic crossover and mutation operators

are replaced with the intelligent path relinking and ejection chain heuristic operators which allow us to design guided spatial moves around the underlying non-linear decision space. These two spatial recombination operators, PRCRX, and ECMUT, satisfy strong spatial constraints such as contiguity and partition and generate new feasible solutions as we chain individual spatial moves together through randomization and optimization on solution fitness. This design allows the development of particular spatial moves and chaining strategies to incorporate domain-specific problem insights in spatial optimization. Research questions **Q5–6** are answered by this work:

Answer to Q5. The disruption of spatial configuration as a result of linear combinations on non-linear decision space in SO is the major limitation of EA effectiveness for SO. An effective spatial EA must leverage the underlying spatial configuration to guide the incremental iterations in searching for new solutions. Meanwhile, the basic principles of crossover and mutation should be followed. EA recombination operators must be not only spatially aware, but also spatially explicit in exploring the spatially constrained decision space.

Answer to Q6. The answer is two-fold. First, linear recombination in EA is replaced by path relinking and ejection chain, two well-known effective heuristics that can provide guided traversal of decision space. A path relinking-based crossover operator is designed to construct a search path between two or more parent solutions, along which intermediate solutions are generated. The path captures search regions travelled during the relinking process. An ejection chain-based mutation operator is designed to conduct neighborhood search from a single parent solution through a chain of local moves. Second, the incorporation of spatial configuration in employing path relinking and ejection chain forms new spatial crossover and mutation operators for effective search on spatially constrained decision space. Various spatial elements such as adjacency, contiguity, containment, compactness, and partition are embedded considerations in local spatial moves, the relinking and the chaining process.

5.1.4 Supporting Grants

Some computational experiments in this thesis used Blue Waters, ROGER, and XSEDE resources that are supported by the National Science Foundation (NSF) under grant numbers: 1238993, 1429699, 1548562, respectively. This thesis is based in part upon work supported by NSF under grant number 1047916. Any opinions, findings, and conclusions or recommen-

dations expressed in this material are those of the author and do not necessarily reflect the views of NSF.

5.2 Discussion and Future Work

My research can be extended in several directions, encompassing computation, algorithm design, and applications.

5.2.1 Search Heuristics on Global Population in PEA

Search strategies that work on local EA population have two major purposes: diversification and intensification. Diversification refers to the search capability of visiting many different regions of the decision space while intensification refers to the ability to obtain high quality solutions within these promising search regions. Diversification and intensification, when applied appropriately during the evolutionary search process for optimization problem-solving, can significantly improve the performance of the search algorithm. Diversification is also essential for statistical applications that require that the identified solutions are independent of one another.

Explicit diversification and intensification strategies in EA can guide the evolutionary process toward extensive and intensive search behaviors that have been leveraged in the current literature for better performance in EA algorithms. Such strategies are not yet supported at the global population level in PEA even though each individual EA process can conduct these strategies by itself. Given the asynchronous communication model adopted in my PEA design, implementing diversification and intensification strategies on the global population will require the design of a series of distributed computing protocols that are also asynchronous and scalable, without the need for global synchronization. An exploration of an efficient version of this protocol and how the global diversification and intensification affects the evolving of both the global and local population will be the topic of future work. The asynchrony raises several interesting research considerations.

First, efficient asynchronous communication protocols at application level needs to be developed to enable collective message passing functions such as broadcasting. I have implemented a distributed broadcast protocol that leverages a spanning tree of the processor topology and achieves a delay less than the topology graph diameter. The impact of using this protocol and the associated communication delays on search performance of PEA needs to be further investigated.

Second, I am looking into how diversification can be leveraged to control the homogeneity of global population. Two features are being developed, accordingly. A global tabu list, comprised of found solutions, directs processor effort, preventing overlapping searches in similar parts of the decision space. Each processor scans the global tabu list and may randomly restart if there is substantial overlap between the tabu list and the local population. The second diversification feature checks the similarity of search regions with other processors once the solution quality of local population exceeds a defined threshold. If processors that receive the check message are searching in similar regions of the decision space, they scatter to another search area.

Third, an intensification strategy is being developed to garner more computing power to climb a peak at a processor that could not make progress in a number of iterations. When a processor sends out an intensification message, a receiving processor probabilistically chooses to provide aid, with probability increasing as iterations without the identification of a new elite increases. If the processor decides to help, the current local population is saved and a new EA search process is conducted on the solution which the sending processor has difficulty to improve. The number of such helpers is controlled as a fraction of the total number of processors and is also implemented in a distributed fashion by flipping a coin locally at each receiving processor.

Lastly, the interactions between diversification and intensification on the entire set of processors need to be investigated to eliminate potential interference and work collectively to achieve better search performance than applying one of the other strategy separately.

5.2.2 Multi-objective Optimization

PEAR is a multi-objective evolutionary algorithm (MOEA) (Zhou et al., 2011; Zitzler, Deb and Thiele, 2000; Coello, Lamont and Veldhuizen, 2006) that uses an aggregation method, i.e., the weighted sum, to combine objectives into a scalar function. The configuration of weights highly depends on domain knowledge and is not desirable for typical decision-making processes where decision makers want to see a set of near optimal solutions showing preferences on different objectives. *Pareto-based fitness assignment* (Deb and Jain, 2014) is a well-known approach to produce a Pareto front that represents a multi-dimensional tradeoff surface with each dimension representing an objective. The optimality of a solution on this surface is determined by Pareto *dominance*, which if solution A dominates solution B, the value of every objective in A is not worse than that in B and at least one objective value is better than B. Pareto-based approaches produce a set of non-dominated solutions,

called a Pareto front which means that no solution in the set dominates another. Since the Pareto front is multi-dimensional, an analysis can be conducted to evaluate solutions that are particularly good on one or more preferred objectives.

Designing an effective Pareto-based MOEA, however, is significantly challenging (Zitzler and Thiele, 1999), a typical effect of the *curse of dimensionality* on the number of objectives:

- The size of nondominated solutions can increase dramatically as more objectives need to be considered;
- Measuring the distance between two nondominated solutions is computationally costly, affecting the performance of neighborhood search;
- EA operators, when applied on multi-dimensional search space, is highly likely to produce child solutions distant from them, compared to the classic one dimensional crossover and mutation.

These challenges, I believe, can be efficiently resolved by using massively parallel MOEA and designing effective EA operators for traversing multi-dimension objective space. A global non-dominated solution set can be distributed on all the processors and is scalable in a massively parallel computing environment. The computation cost for measuring the distance among solutions in this set can also be distributed and coordinated through the design of a distributed message passing protocol. This protocol, like the protocols used in global population, is asynchronous for high efficiency. With this computational approach, I can design further strategies to consider the extent of each objective in the global nondominated set such that the Pareto front can represent a comprehensive distribution of nondominated solutions on all objectives specified by decision makers for consideration. I expect that a set of specialized EA operators will be designed to handle multiple objectives in spatial optimization because interactions among the dimensions in the objective space are also highly influenced by spatial elements in the optimization problem. The outcome of this research will be a massively parallel MOEA. Using this solution for political redistricting will greatly enhance PEAR's applicability in practical redistricting scenarios because in a redistricting law suit, conflicting objectives that represent multiple dimensions of interest (e.g., of minority rights, partisanship, race, etc.) are common. As these objectives are proposed and debated between plaintiff and defendant, court needs a quantitative interpretation of a proposed solution that gives clear explanation on the preferences represented by this solution such that a meaningful comparison among multiple solutions can be made for judgement. A Pareto

solution set is also useful for statistical analysis because it provides a way to numerically correlate a solution to multiple objectives as variates.

5.2.3 Enhancing Spatial Element Handling

The design of SEEC considered a few spatial elements: adjacency, contiguity, partition, topological relations such as containment, shape (compactness). More sophisticated constraints are being explored, including patterns and spatial uncertainty. Topological and geometric calculations may be necessary to be performed, but are computationally expensive. Efficient implementation in SEEC thus becomes critical. A SEEC implementation with a comprehensive set of spatial optimization characteristics will open doors to a broad spectrum of spatial optimization applications. I plan to pursue a software approach that has the potential to transform the practice of EA usage, which is currently limited in exposing spatial element handling details to end users. An explicit representation of spatial elements in problem formulation, interpretation of solutions, and controlling EA parameters and search strategies will significantly accelerate the process of injecting problem knowledge into the problem solving process and help users understand the spatial footprint of the solution evolving trajectory.

5.2.4 Software

SEEC provides a general spatial evolutionary computation framework for SO applications. I plan to release SEEC as an EA solver for SO in the near future.

SEEC will expose general EA functions as most of EA solvers do, including problem encoding, initial solution generation, selection, crossover and mutation, replacement, and stopping criteria. Abstract interface will be provided as score function definition for fitness evaluation. Penalty mechanism will be provided. Users have the flexibility to use penalty function to guide the search in feasible region separately or mixing it in objective functions. For efficiency purpose, an update interface will be provided for fitness and penalty functions, instead of evaluating the fitness and unfitness from scratch. Users will be responsible on the implementation of update functions, for example, using dynamic programming techniques. The crossover and mutation implementation includes both aspatial and spatial path relinking and ejection chain.

Spatial element handling is embedded in SEEC and can be configured. Currently, efficient implementation of distance, adjacency, and contiguity is supported. For spatial elements not supported yet or to be handled using user-provided functions, users have the option to add or

override existing handling functions. SEEC integrates GDAL for geospatial processing. Users can directly use GEOS/GDAL function bindings for topological and geometric operations. The SEEC application programming interface (API) will be provided as a C++ library with multiple language bindings such as Python.

In complex modeling and decision-making applications, the SEEC solver can be used as a building block to approximate global optimal solutions. As EA is known to be robust for nonlinear non-convex problems, it can be used for a broad range of optimization problems. However, problem size directly affects SEEC's performance within a reasonable amount of time. Variable reduction or fixing techniques should be used when possible before the problem encoding phase. For example, dependent continuous variables in equality constraints can be identified and substituted (Boulos, Lansey and Karney, 2004; Housh et al., 2015). Good starting population is also critical for an EA's performance in a timely fashion. A piece-by-piece domain decomposition approach developed in Cai, McKinney and Lasdon (2001*a*) and Cai (2008) can be used to iteratively improving starting population quality, following a defined domain sequence. In this approach, a domain independently defines its decision variables, objectives, and constraints. Each iteration of this approach calls a solver to optimize a selected domain. The output solution then serves as the initial fixed values for that domain in the next iteration which optimizes another domain. This approach is suitable for solving large nonlinear problems. SEEC, as an EA solver, would be a building block in such holistic modeling approaches.

5.2.5 Open Platform for Redistricting Research

In the substantive realm of political redistricting, computational models such as the one developed in PEAR, allow us to synthesize and organize massive amounts of computation and data to evaluate redistricting schemes and tailor them to notions of "fairness" and democratic rule. Importantly, our model is deeply rooted in the long-standing values and goals for democracy in the U.S. as they has been outlined by the Supreme Court. Our optimization algorithm is not borne out of convenience and ease, but instead, strongly tailored to the existing social systems and their structures and mechanisms for securing fairness and transparency in the electoral systems.

While our computational approach is formulated via an optimization strategy, our goal is not to seek a perfectly optimized plan per se because the perfectly optimized plan, while interesting, is not necessary, and further, has only limited value. Plainly, it is difficult, and likely impossible to identify the optimal plan given the many competing interests in any

redistricting effort. No single plan will satisfy every interested stakeholder—there is no perfect plan. If there is no perfect plan for every constituent, then it makes sense that we will instead be choosing from a bounded set of "reasonably imperfect plans" (Cain, 2012). The discrete optimization framework is ideal as a vehicle for identifying large sets of "reasonably imperfect" redistricting plans. The set of reasonably imperfect plans is useful at the districting drawing stage as well as for judges who are adjudicating the constitutionality of a redistricting plan. At the drawing stage, this information suitably drives an iterative bargaining process whether that process involves partisan legislators or members of an independent redistricting commission. When this bounded set of plans can be identified and the elements of the set have associated indicators of partisan bias, responsiveness, efficiency gaps, compactness, respect for political subdivisions, etc., the redistricting process is imbued with valuable structure that is otherwise non-existent. That structure alone makes the redistricting process more transparent and may itself serve to reduce legal challenges. When a legal challenge is mounted, this large set of reasonably imperfect plans produces the relevant background and allows one to place and understand the proposed plan in context. The purpose of our optimization strategy, then, is to search, synthesize, and organize massive amounts of information that will supply a common base of knowledge to guide an informed and intelligent debate.

REFERENCES

Agapakis, Christina M., Patrick M. Boyle and Pamela A. Silver. 2012. "Natural Strategies for the Spatial Optimization of Metabolism in Synthetic Biology." *Nature Chemical Biology* 8:527–535.

Alba, E, G Luque and J.M Troya. 2004. "Parallel LAN/WAN heuristics for optimization." *Parallel Computing* 30(56):611 – 628.

Alba, E. and J. M. Troya. 2002. "Improving flexibility and efficiency by adding parallelism to genetic algorithms." *Statistics and Computing* 12(2):91–114.

Alba, E. and M. Tomassini. 2002. "Parallelism and evolutionary algorithms." *IEEE Transactions on Evolutionary Computation* 6(5):443–462.

Alba, Enrique and Jose M Troya. 1999*a*. An Analysis of Synchronous and Asynchronous Parallel Distributed Genetic Algorithms with Structured and Panmictic Islands. In *Proceedings of the IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*. London, UK: Springer-Verlag pp. 248–256.

Alba, Enrique and José M. Troya. 1999*b*. "A survey of parallel distributed genetic algorithms." *Complexity* 4(4):31–52.

Altman, Micah and Michael P. McDonald. 2011. "BARD: Better Automated Redistricting." *Journal of Statistical Software* 42(4):1–28.

Amini, Mohammad M. and Michael Racer. 1994. "A Rigorous Computational Comparison of Alternative Solution Methods for the Generalized Assignment Problem." *Management Science* 40(7):pp. 868–890.

Andre, David and John R. Koza. 1996. "Parallel genetic programming: a scalable implementation using the transputer network architecture." pp. 317–337.

Asanovic, Krste, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams and Katherine A. Yelick. 2006. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183 EECS Department, University of California, Berkeley.

Ashby, Steve, Pete Beckman, Jackie Chen, Phil Colella, Bill Collins, Dona Crawford, Jack Dongarra, Doug Kothe, Rusty Lusk, Paul Messina et al. 2010. "The opportunities and challenges of exascale computing." *Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee* pp. 1–77.

Bader, David. 2007. *Petascale Computing: Algorithms and Applications*. 1 ed. Chapman & Hall/CRC.

Balachandran, V. 1976. "An Integer Generalized Transportation Model for Optimal Job Assignment in Computer Networks." *Operations Research* 24(4):pp. 742–759.

Baluja, Shumeet. 1992. A Massively Distributed Parallel Genetic Algorithm (mdpGA). Technical report Carnegie Mellon University.

Belding, Theodore C. 1995. The Distributed Genetic Algorithm Revisited. In *Proceedings of the 6th International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. pp. 114–121.

Borndorfer, R. and R. Weismantel. 1997. "Relations among some combinatorial programs.".

Boulos, PF, KE Lansey and BW Karney. 2004. "Comprehensive water distribution systems analysis handbook for engineers and planners. MWH Soft." *Inc. Pasadena, CA* .

BoussaïD, Ilhem, Julien Lepagnot and Patrick Siarry. 2013. "A survey on optimization metaheuristics." *Information Sciences* 237:82–117.

Brookes, Christopher J. 2001. "A genetic algorithm for designing optimal patch configurations in GIS." *International Journal of Geographical Information Science* 15(6):539–559.

Brookes, C.J. 1997. "A Parameterized Region-Growing Programme for Site Allocation on Raster Suitability Maps." *International Journal of Geographical Information Science* 11:375–396.

Cai, Ximing. 2008. "Implementation of Holistic Water Resources-economic Optimization Models for River Basin Management - Reflective Experiences." *Environ. Model. Softw.* 23(1):2–18.

Cai, Ximing, Daene C. McKinney and Leon S. Lasdon. 2001*a*. "Piece-by-Piece Approach to Solving Large Nonlinear Water Resources Management Models." *Journal of Water Resources Planning and Management* 127(6):363–368.

Cai, Ximing, Daene C McKinney and Leon S Lasdon. 2001*b*. "Solving nonlinear water management models using a combined genetic algorithm and linear programming approach." *Advances in Water Resources* 24(6):667–676.

Cain, Bruce E. 2012. "Redistricting Commissions: A Better Political Buffer?" *Yale Law Journal* 121(7):1808–1844.

Cantu-Paz, E. 1997. "A Survey of Parallel Genetic Algorithms.".

Cantu-Paz, Erick and David E Goldberg. 2000. "Efficient parallel genetic algorithms: theory and practice." *Computer methods in applied mechanics and engineering* 186(2):221–238.

Cattrysse, Dirk G. and Luk N. Van Wassenhove. 1992. "A survey of algorithms for the generalized assignment problem." *European Journal of Operational Research* 60(3):260 – 272.

Chekuri, Chandra and Sanjeev Khanna. 2000. A PTAS for the Multiple Knapsack Problem. In *SODA 2000: Proceedings of the 11th Annual Symposium on Discrete Algorithms*. pp. 213–222.

Chen, Hao, Nicholas S. Flann and Daniel W. Watson. 1998. "Parallel Genetic Simulated Annealing: A Massively Parallel SIMD Algorithm." *IEEE Trans. Parallel Distrib. Syst.* 9(2):126–136.

Chen, Jowei and Jonathan Rodden. 2013. "Unintentional Gerrymandering: Political Geography and Electoral Bias in Legislatures." *Quarterly Journal of Political Science* 8:239–269.

Cho, Wendy K. Tam and Yan Y. Liu. 2016. "Toward a Talismanic Redistricting Tool: A Computational Method for Identifying Extreme Redistricting Plans." *Election Law Journal* 15:351–366.
URL: *http://online.liebertpub.com/doi/abs/10.1089/elj.2016.0384*

Chu, P. C. and J. E. Beasley. 1997. "A genetic algorithm for the generalised assignment problem." *Comput. Oper. Res.* 24(1):17–23.

Church, Richard L. 1999. Location Modelling and GIS. In *Geographical Information Systems & Science, 2e*, ed. Paul A. Longley, Michael F. Goodchild, David J. Maguire and David W. Rhind. Wiley pp. 293–303.

Church, Richard L. and Charles S. ReVelle. 1974. "The Maximal Covering Location Problem." *Papers of the Regional Science Association* 32:101–118.

Church, Richard L and Thomas J Cova. 2000. "Mapping evacuation risk on transportation networks using a spatial optimization model." *Transportation Research Part C: Emerging Technologies* 8(1):321–336.

Cirincione, Carmen, Thomas A. Darling and Timothy G. O'Rourke. 2000. "Assessing South Carolina's 1990s Congressional Districting." *Political Geography* 19(2):189–211.

Clausen, Jonathan R., Daniel A. Reasor and Cyrus K. Aidun. 2010. "Parallel performance of a lattice-Boltzmann/finite element cellular blood flow solver on the IBM Blue Gene/P architecture." *Computer Physics Communications* 181(6):1013 – 1020.

Cledat, Romain, Tushar Kumar, Jaswanth Sreeram and Santosh Pande. 2009. Opportunistic Computing: A New Paradigm for Scalable Realism on Many-Cores. In *HOTPAR '09: USENIX Workshop on Hot Topics in Parallelism*.

Coello, Carlos A. Coello, Gary B. Lamont and David A. Van Veldhuizen. 2006. *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition*. 3rd ed. The MIT Press.

Cova, Thomas J and Richard L Church. 2000. "Contiguity constraints for single-region site search problems." *Geographical Analysis* 32(4):306–329.

Craig, Samuel, Avijit Ghosh and Sara L McLafferty. 1984. "Models of Retail Location Process: A Review." *Journal of Retailing* 60(1):5–36.

Cromley, Robert G. and Dean M. Hanink. 1999. "Coupling land use allocation models with raster GIS." *Journal of Geographical Systems* 1:137–153.

Davis, Lawrence. 1989. Adapting operator probabilities in genetic algorithms. In *proc. 3rd International conference on genetic algorithms*. pp. 61–69.

Deb, K. and H. Jain. 2014. "An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints." *IEEE Transactions on Evolutionary Computation* 18(4):577–601.

Diaz, Juan A. and Elena Fernandez. 2001. "A Tabu search heuristic for the generalized assignment problem." *European Journal of Operational Research* 132(1):22–38.

Dongarra, Jack, Dennis Gannon, Geoffrey Fox and Ken Kennedy. 2007. "The Impact of Multicore on Computational Science Software." *CTWatch Quarterly* 3(1):817–831.

Downs, Roger M et al. 2006. *Learning to think spatially*. National Academies Press.

Duczmal, Luiz, Andr L.F. Canado, Ricardo H.C. Takahashi and Luprcio F. Bessegato. 2007. "A genetic algorithm for irregularly shaped spatial scan statistics." *Computational Statistics and Data Analysis* 52(1):43 – 52.

Duque, Juan C., Richard L. Church and Richard S. Middleton. 2011. "The p-Regions Problem." *Geographical Analysis* 43(1):104–126.
URL: *http://dx.doi.org/10.1111/j.1538-4632.2010.00810.x*

DAmbrosio, D. and W. Spataro. 2007. "Parallel evolutionary modelling of geological processes." *Parallel Computing* 33(3):186 – 212.

Eklund, Sven E. 2004. "A massively parallel architecture for distributed genetic algorithms." *Parallel Computing* 30(56):647 – 676.

Farahani, Reza Zanjirani, Nasrin Asgari, Nooshin Heidari, Mahtab Hosseininia and Mark Goh. 2012. "Covering problems in facility location: A review." *Computers & Industrial Engineering* 62(1):368–407.

Faraj, Ahmad, Pitch Patarasuk and Xin Yuan. 2007. A study of process arrival patterns for MPI collective operations. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*. New York, NY, USA: ACM pp. 168–179.

Feltl, Harald and Gunther R. Raidl. 2004. An improved hybrid genetic algorithm for the generalized assignment problem. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*. New York, NY, USA: ACM pp. 990–995.

Feo, Thomas A. and Mauricio G. C. Resende. 1995. "Greedy Randomized Adaptive Search Procedures." *Journal of Global Optimization* 6:109–133.

Fisher, Marshall L., R. Jaikumar and Luk N. Van Wassenhove. 1986. "A Multiplier Adjustment Method for the Generalized Assignment Problem." *Management Science* 32(9):pp. 1095–1103.

Fleischer, M.A. 1995. "Simulated Annealing: Past, Present, and Future." *Proceedings of the 27th Winter Simulation Conference* 00:155–161.

Fogel, David B. 1997. "Evolutionary algorithms in theory and practice." *Complexity* 2(4):26–27.

Folino, G, C Pizzuti and G Spezzano. 2003. "A scalable cellular implementation of parallel genetic programming." *IEEE Transactions on Evolutionary Computation* 7(1):37–53.

Francis, Richard L. and John A. White. 1974. *Facility Layout and Location: An Analytical Approach*. Englewood Cliffs, NJ: Prentice Hall.

French, Alan P. and John M. Wilson. 2007. "An LP-based heuristic procedure for the generalized assignment problem with special ordered sets." *Comput. Oper. Res.* 34(8):2359–2369.

Freville, Arnaud. 2004. "The multidimensional 0-1 knapsack problem: An overview." *European Journal of Operational Research* 155(1):1–21.

Frolov, Y.S. 1975. "Measuring Shape of Geographical Phenomena: A History of the Issue." *Soviet Geography Review and Translation* 16(10):676–687frolov.

Galinier, Philippe and Jin-Kao Hao. 1999. "Hybrid Evolutionary Algorithms for Graph Coloring." *Journal of Combinatorial Optimization* 3(4):379–397.

Garey, M.R. and D.S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W.H. Freeman and Co.

Garfinkel, R. S. and G. L. Nemhauser. 1970. "Optimal Political Districting by Implicit Enumeration Techniques." *Management Science* 16(8):B–495–B–508.

Gearhart, Burton C. and John M. Liittschwager. 1969. "Legislative Districting by Computer." *Behavioral Science* 14(5):404–417.

Glover, Fred. 1994. "Genetic algorithms and scatter search: unsuspected potentials." *Statistics and Computing* 4(2):131–140.
URL: *http://dx.doi.org/10.1007/BF00175357*

Glover, Fred and Manuel Laguna. 1997. *Tabu Search*. Boston, MA: Kluwer Academic Publishers.

Glover, Fred, Manuel Laguna and Rafael Marti. 2000. "Fundamentals of Scatter Search and Path Relinking." *Control and Cybernetics* 29(3):653–684.

Goldberg, David. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley Professional.

Goldberg, David E. and Kalyanmoy Deb. 1991. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In *Foundations of Genetic Algorithms*, ed. Gregory J. E. Rawlins. San Francisco, CA: Morgan Kaufmann pp. 69–93.

Goodchild, Michael F. and Donald G. Janelle. 2010. "Toward critical spatial thinking in the social sciences and humanities." *GeoJournal* 75(1):3–13.

Gordon, V. Scott and L. Darrell Whitley. 1993. Serial and Parallel Genetic Algorithms as Function Optimizers. In *Proceedings of the 5th International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. pp. 177–183.

Grofman, Bernard. 1982. For Single Member Districts Random is not Equal. In *Representation and Redistricting Issues*, ed. Bernard Grofman, Arend Lijphart, Robert B. McKay and Howard A. Scarrow. Lexington Books.

Gudgin, G. and P. J. Taylor. 1979. *Seats, Votes and the Spatial Organization of Elections*. London: Pion.

Guo, Diansheng. 2008. "Regionalization with dynamically constrained agglomerative clustering and partitioning (REDCAP)." *International Journal of Geographical Information Science* 22(7):801–823.

Hamerly, Greg and Charles Elkan. 2002. Alternatives to the K-means Algorithm That Find Better Clusterings. In *Proceedings of the Eleventh International Conference on Information and Knowledge Management*. CIKM '02 New York, NY, USA: ACM pp. 600–607.

Harris, Curtis. 1964. "A Scientific Method of Districting." *Behavioral Science* 9:219–225.

Hart, William E., Scott B. Baden, Richard K. Belew and Scott R. Kohn. 1996. Analysis of the Numerical Effects of Parallelism on a Parallel Genetic Algorithm. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*. Washington, DC, USA: IEEE Computer Society pp. 606–612.

Helsgaun, Keld. 2000. "An effective implementation of the LinKernighan traveling salesman heuristic." *European Journal of Operational Research* 126(1):106 – 130.

Hidalgo, J. Ignacio, Francisco Fernandez, Juan Lanchares, Erick Cant-Paz and Albert Zomaya. 2010. "Parallel Architectures and Bioinspired Algorithms." *Parallel Computing* 36(1011):553 – 554.

Hof, John G. and Michael Bevers. 1998. *Spatial Optimization for Managed Ecosystems*. New York: Columbia University Press.

Holland, John H. 1992. *Adaptation in natural and artificial systems*. Cambridge, MA, USA: MIT Press.

Housh, Mashor, Ximing Cai, Tze Ling Ng, Gregory F. McIsaac, Yanfeng Ouyang, Madhu Khanna, Murugesu Sivapalan, Atul K. Jain, Steven Eckhoff, Stephen Gasteyer, Imad Al-Qadi, Yun Bai, Mary A. Yaeger, Shaochun Ma and Yang Song. 2015. "System of Systems Model for Analysis of Biofuel Development." *Journal of Infrastructure Systems* 21(3):04014050.

Huang, Chun-Hsi and Sanguthevar Rajasekaran. 2004. "High-performance parallel biocomputing." *Parallel Computing* 30(910):999 – 1000.

IBM. 2013. "CPLEX." http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.
URL: *http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/*

Izakian, Hesam and Witold Pedrycz. 2012. "A new PSO-optimized geometry of spatial and spatio-temporal scan statistics for disease outbreak detection." *Swarm and Evolutionary Computation* 4:1 – 11.

Jeet, V. and E. Kutanoglu. 2007. "Lagrangian relaxation guided problem space search heuristics for generalized assignment problems." *European Journal of Operational Research* 127(3):1039–1056.

Jiang, Weihang, Jiuxing Liu, Hyun-Wook Jin, D.K. Panda, W. Gropp and R. Thakur. 2004. "High performance MPI-2 one-sided communication over InfiniBand." *Cluster Computing and the Grid, IEEE International Symposium on* 0:531–538.

Juille, Hugues and Jordan B. Pollack. 1996. Co-evolving Intertwined Spirals. In *in Proceedings of the Fifth Annual Conference on Evolutionary Programming*. MIT Press pp. 461–468.

Kalinowski, T. 1994. Solving the mapping problem with a genetic algorithm on the MasPar-1. In *Massively Parallel Computing Systems, 1994., Proceedings of the First International Conference on*. pp. 370 –374.

Keane, M. 1975. "The Size of the Region-Building Problem." *Environment & Planning A* 7:575–577.

Kent, Brian, B. Bruce Bare, Richard C. Field and Gordon A. Bradley. 1991. "Natural Resource Land Management Planning Using Large-Scale Linear Programming: The USDA Forest Service Experience with FORPLAN." *Operations Research* 39(1):13–27.

King, Douglas M., Sheldon H. Jacobson, Edward C. Sewell and Wendy K. Tam Cho. 2012. "Geo-Graphs: An Efficient Model for Enforcing Contiguity and Hole Constraints in Planar Graph Partitioning." *Operations Research* 60(5):1213–1228.

Konfrst, Zdenek. 2004. "Parallel genetic algorithms: advances, computing trends, applications and perspectives." *Parallel and Distributed Processing Symposium, International* 7:162b.

Lau, T.L. and E.P.K. Tsang. 10-12 Nov 1998. "The guided genetic algorithm and its application to the generalized assignment problem." *Tools with Artificial Intelligence, 1998. Proceedings. Tenth IEEE International Conference on* pp. 336–343.

Li, Wenwen, Michael F. Goodchild and Richard Church. 2013. "An Efficient Measure of Compactness for Two-Dimensional Shapes and its Application in Regionalization Problems." *International Journal of Geographical Information Science* 27(6):1227–1250.

Li, Wenwen, Richard L. Church and Michael F. Goodchild. 2014. "The p-Compact-regions Problem." *Geographical Analysis* 46(3):250–273.

Ligmann-Zielinska, Arika, Richard L Church and Piotr Jankowski. 2008. "Spatial optimization as a generative technique for sustainable multiobjective land-use allocation." *International Journal of Geographical Information Science* 22(6):601–622.

Lin, Shyh-Chang, III Punch, W.F. and E.D. Goodman. 1994. "Coarse-grain parallel genetic algorithms: categorization and new approach." *Parallel and Distributed Processing, 1994. Proceedings. Sixth IEEE Symposium on* pp. 28–37.

Liu, Yan Y. and Shaowen Wang. 2015. "A scalable parallel genetic algorithm for the Generalized Assignment Problem." *Parallel Computing* 46:98–119.
URL: *http://www.sciencedirect.com/science/article/pii/S0167819114000519*

Liu, Yan Y., Wendy K. Tam Cho and Shaowen Wang. 2015. A Scalable Computational Approach to Political Redistricting Optimization. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*. XSEDE '15 New York, NY, USA: ACM pp. 1–2.
URL: *http://doi.acm.org/10.1145/2792745.2792751*

Liu, Yan Y., Wendy K. Tam Cho and Shaowen Wang. 2016. "PEAR: a massively parallel evolutionary computation approach for political redistricting optimization and analysis."

Swarm and Evolutionary Computation 30:78 – 92.
URL: *http://www.sciencedirect.com/science/article/pii/S2210650216300220*

Lorena, L., M. Narciso and J. Beasley. 1999. "A constructive genetic algorithm for the generalized assignment problem." *Evolutionary Optimization* .

Lucas, Robert, J Ang, K Bergman, S Borkar, W Carlson, L Carrington, G Chiu, R Colwell, W Dally, J Dongarra et al. 2014. "Top ten exascale research challenges." *DOE ASCAC subcommittee report* pp. 1–86.

Macmillan, William and Todd Pierce. 1994. Optimization Modelling in a GIS Framework: The Problem of Political Districting. In *Spatial Analysis and GIS*, ed. Stewart Fotheringham and Peter Rogerson. Taylor & Francis chapter 11, pp. 221–246.

Manderick, B. and P. Spiessens. 1989. Fine-grained parallel genetic algorithms. In *Proceedings of the third international conference on Genetic algorithms*. Morgan Kaufmann Publishers Inc. pp. 428–433.

Mascagni, Michael and Ashok Srinivasan. 2000. "Algorithm 806: SPRNG: a scalable library for pseudorandom number generation." *ACM Trans. Math. Softw.* 26(3):436–461.

McCool, M.D. 2008. "Scalable Programming Models for Massively Multicore Processors." *Proceedings of the IEEE* 96(5):816–831.

McLafferty, Sara L and Avijit Ghosh. 1986. "Multipurpose shopping and the location of retail firms." *Geographical Analysis* 18(3):215–226.

Mehrotra, Anuj, Ellis L. Johnson and George L. Nemhauser. 1998. "An Optimization Based Heuristic for Political Districting." *Management Science* 44(8):1100–1114.

Mezura-Montes, Efren and Carlos A. Coello Coello. 2011. "Constraint-handling in nature-inspired numerical optimization: Past, present and future." *Swarm and Evolutionary Computation* 1(4):173 – 194.

Michalewicz, Zbigniew and Cezary Z Janikow. 1991. "Genetic algorithms for numerical optimization." *Statistics and Computing* 1(2):75–91.

Minor, S.D. and T.L. Jacobs. 1994. "Optimal Land Allocation for Solid and Hazadous Waste Landfill Siting." *Journal of Envionmental Engineering* 120:1095–1108.

MPI-Forum. 2012. *MPI: A Message-Passing Interface Standard, Version 3.0*. Stuttgart, Germany: High Performance Computing Center Stuttgart (HLRS).

Muraro, Daniele and Rui Dilao. 2013. "A parallel multi-objective optimization algorithm for the calibration of mathematical models." *Swarm and Evolutionary Computation* 8:13 – 25.

Murray, Alan T. 2016. "Maximal Coverage Location Problem Impacts, Significance, and Evolution." *International Regional Science Review* 39(1):5–27.

Murray, Alan T, Morton E O'Kelly and Richard L Church. 2008. "Regional service coverage modeling." *Computers & Operations Research* 35(2):339–355.

Murray, Alan T., Tony H. Grubesic and Ran Wei. 2014. "Spatially significant cluster detection." *Spatial Statistics* 10(Supplement C):103–116.

Nagel, Stuart S. 1965. "Simplified Bipartisan Computer Redistricting." *Stanford Law Review* 17(5):863–899.

Nemhauser, G.L. and L.A. Woolsey. 1988. *Integer and Combinatorial Optimization*. New York: John Wiley & Sons.

Ngo, J. Thomas and Joe Marks. 1993. "Physically Realistic Motion Synthesis in Animation." *Evolutionary Computation* 1(3):235–268.

Ocenasek, Jiri and Martin Pelikan. 2004. Parallel mixed Bayesian optimization algorithm: A scaleup analysis. In *In S. Cagnoni (Ed.), Workshop Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004). Electronic publication.*

O'Kelly, Morton E and Harvey J. Miller. 1994. "The Hub Network Design Problem: A Review and Synthesis." *Journal of Transportation Geography* 2(1):31–40.

Oliveto, Pietro S., Jun He and Xin Yao. 2007. "Time complexity of evolutionary algorithms for combinatorial optimization: a decade of result." *International Journal of Automation and Computing* 4(3):281–293.

Openshaw, S and L Rao. 1995. "Algorithms for Reengineering 1991 Census Geography." *Environment and Planning A* 27(3):425–446.

Osserman, Robert. 1978. "Isoperimetric Inequality." *Bulletin of the American Mathematical Society* 84(6):1182–1238.

Papayanopoulos, L. 1973. "Quantitative Principles Underlying Apportionment Methods." *Annals of the New York Academy of Sciences* 219:3–4.

Patvardhan, C., Sulabh Bansal and A. Srivastav. In press. "Parallel improved quantum inspired evolutionary algorithm to solve large size Quadratic Knapsack Problems." *Swarm and Evolutionary Computation* .

Prabhu, Devaraya, Bill P. Buckles and Frederick E. Petry. 2006. "A SIMD Environment for Genetic Algorithms with Interconnected Subpopulations." *Scalable Computing: Practice and Experience* 7(2):65–86.

Qian, Fubin and Rui Ding. 2007. "Simulated Annealing for the 01 Multidimensional Knapsack Problem." *Journal of Chinese Universities Numerical Mathematics* 16(4):320–327.

Rivera, Wilson. 2001. "Scalable Parallel Genetic Algorithms." *Artif. Intell. Rev.* 16(2):153–168.

Rossiter, David J. and Ron J. Johnston. 1981. "Program GROUP: The Identification of All Possible Solutions to a Constituency-Delimitation Problem." *Environment and Planning A* 13(2):231–238.

Ruciski, M., D. Izzo and F. Biscani. 2010. "On the impact of the migration topology on the Island Model." *Parallel Computing* 36(1011):555 – 571.

Sarkar, Vivek, William Harrod and Allan Snavely. 2009. "Software challenges in extreme scale systems." *Journal of Physics Conference Series* 180(1):012–045.

Schilling, David A., Vaidyanathan Jayaraman and Reza Barkhi. 1993. "A Review of Covering Problems in Facility Location." *Location Science* 1:25–55.

Shapiro, Bruce A., Jin Chu Wu and David Bengali. February 2001. "The massively parallel genetic algorithm for RNA folding: MIMD implementation and population variation." *Bioinformatics* 17(2):137–148.

Shepherd, J. W. and M. A. Jenkins. 1970. "Decentralizing High School Administration in Detroit: A Computer Evaluation of Alternative Strategies of Political Control." *Economic Geography* 48(1):95–106.

Shirabe, Takeshi. 2005. "A model of contiguity for spatial unit allocation." *Geographical Analysis* 37(1):2–16.

Sun, Xian-He and Lionel M. Ni. 1990. Another view on parallel speedup. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. Supercomputing '90 Los Alamitos, CA, USA: IEEE Computer Society Press pp. 324–333.
URL: *http://dl.acm.org/citation.cfm?id=110382.110450*

Tanese, Reiko. 1989. Distributed Genetic Algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. pp. 434–439.

Tayarani-N, Mohammad-H and Adam Prügel-Bennett. 2014. "On the landscape of combinatorial optimization problems." *IEEE Transactions on Evolutionary Computation* 18(3):420–434.

Thoreson, James and John Liittschwager. 1967. "Computers in Behavioral Science: Legislative Districting by Computer Simulation." *Behavioral Science* 12:237–247.

Tong, Daoqin, Alan Murray and Ningchuan Xiao. 2009. "Heuristics in spatial analysis: a genetic algorithm for coverage maximization." *Annals of the Association of American Geographers* 99(4):698–711.

Tong, Daoqin and Alan T Murray. 2012. "Spatial optimization in geography." *Annals of the Association of American Geographers* 102(6):1290–1309.

Vickrey, William S. 1961. "On the Prevention of Gerrymandering." *Political Science Quarterly* 76:105–110.

Von Thunen, Johann Heinrich. 1842. Der Isolierte Staat. In *Von Thünens Isolated State*. London: Pergamon.

Wang, Fahui, Diansheng Guo and Sara McLafferty. 2012. "Constructing geographic areas for cancer data analysis: A case study on late-stage breast cancer risk in Illinois." *Applied Geography* 35(1):1 – 11.

Wang, Jihua. 2013. Solving large-scale spatial optimization problems in groundwater management PhD thesis University of Illinois at Urbana-Champaign.
URL: *https://search.proquest.com/docview/1467480799?accountid=14553*

Weaver, J. B. and S. W. Hess. 1963. "A Procedure for Nonpartisan Districting: Development of Computer Techniques." *Yale Law Journal* 72:288–308.

Wei, Ran and Alan T Murray. 2013. "A multi-objective evolutionary algorithm for facility dispersion under conditions of spatial uncertainty." *Journal of the Operational Research Society* 65(7):1133–1142.

Whitley, Darrell. 2001. "An overview of evolutionary algorithms: practical issues and common pitfalls." *Information and Software Technology* 43(14):817–831.

Whitley, L. Darrell. 1993. Cellular Genetic Algorithms. In *Proceedings of the 5th International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. pp. 658–.

Williams, Justin C. 2002. "A Zero-One Programming Model for Contiguous Land Acquisition." *Geographical Analysis* 34(4):330–349.

Wilson, J. M. 1997. "A Genetic Algorithm for the Generalised Assignment Problem." *The Journal of the Operational Research Society* 48(8):804–809.

Wright, Alden H et al. 1991. "Genetic algorithms for real parameter optimization." *Foundations of genetic algorithms* 1:205–218.

Wright, J.C., C.S. ReVelle and J. Cohon. 1983. "A Multiobjective Integer Programming Model for the Land Acquisition Problem." *Regional Science and Urban Economics* 13:31–53.

Wright, Sewall. 1932. The roles of mutation, inbreeding, crossbreeding and selection in evolution. In *Proc. 6th Int. Cong. Genet.* Vol. 1 pp. 356–366.

Wu, Xiaolan and Alan T Murray. 2008. "A new approach to quantifying spatial contiguity using graph theory and spatial interaction." *International Journal of Geographical Information Science* 22(4):387–407.

Wu, Xiaolan, Alan T Murray and Ningchuan Xiao. 2011. "A multiobjective evolutionary algorithm for optimizing spatial contiguity in reserve network design." *Landscape ecology* 26(3):425–437.

Xiao, Ningchuan. 2008. "A Unified Conceptual Framework for Geographical Optimization Using Evolutionary Algorithms." *Annals of the Association of American Geographers* 98(4):795–817.

Xiao, Ningchuan, David A Bennett and Marc P Armstrong. 2002. "Using Evolutionary Algorithms to Generate Alternatives for Multiobjective Site-Search Problems." *Environment and Planning A* 34(4):639–656.
URL: *http://epn.sagepub.com/content/34/4/639.abstract*

XSEDE. 2017. http://xsede.org.
URL: *http://xsede.org*

Yagiura, Mutsunori, Toshihide Ibaraki and Fred Glover. 2004. "An Ejection Chain Approach for the Generalized Assignment Problem." *INFORMS Journal on Computing* 16(2):133–151.

Yagiura, Mutsunori, Toshihide Ibaraki and Fred Glover. 2006. "A path relinking approach with ejection chains for the generalized assignment problem." *European Journal of Operational Research* 127(2):548–569.

Zhou, A., B.-Y. Qu, H. Li, S.-Z. Zhao, P.N. Suganthan and Q. Zhang. 2011. "Multiobjective evolutionary algorithms: A survey of the state of the art." *Swarm and Evolutionary Computation* 1(1):32–49. cited By 350.

Zitzler, E. and L. Thiele. 1999. "Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach." *IEEE Transactions on Evolutionary Computation* 3(4):257–271.

Zitzler, Eckart, Kalyanmoy Deb and Lothar Thiele. 2000. "Comparison of Multiobjective Evolutionary Algorithms: Empirical Results." *Evolutionary Computation Journal* 8(2):173–195.
URL: *http://dx.doi.org/10.1162/106365600568202*